AFRL-IF-RS-TR-2004-71
Final Technical Report
March 2004

INFORMATION ASSURANCE SCIENCE AND ENGINEERING PROJECT

Carnegie-Mellon University

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2004-71 has been reviewed and is approved for publication.

APPROVED: /s/
CHARLES P. SATTERTHWAITE
Project Engineer

FOR THE DIRECTOR:

/s/
JAMES A COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 2004	Final Apr 00 – Apr 02

4. TITLE AND SUBTITLE

INFORMATION ASSURANCE SCIENCE AND ENGINEERING PROJECT

5. FUNDING NUMBERS

C - F30602-00-2-0523
PE - 63760E
PR - IAST
TA - 00
WU - 09

6. AUTHOR(S)

Tom Longstaff and Jeannette Wing

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Carnegie-Mellon University
Pittsburgh, PA 15312

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFT
26 Electronic Pky
Rome NY 13441-4514

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2004-71

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Charles P. Satterthwaite, AFRL/IFTA (Wright Site), 937-255-6548, x3584

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Creation of metrics, methodologies, and tools for the implementation of assurance in information system design and assessment processes. Develop a scientific framework for understanding and developing information assurance systems and for reasoning about the assurance aspects of these systems.

14. SUBJECT TERMS

Command and control, information assurance, knowledge engineering, robust systems, secure systems

15. NUMBER OF PAGES

80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

Abstract

An integral part of modeling the global view of network security is constructing attack graphs*. In practice, attack graphs are produced manually by Red Teams. Construction by hand, however, is tedious, error-prone, and impractical for attack graphs larger than a hundred nodes. In this paper we present an automated technique for generating and analyzing attack graphs. We base our technique on* symbolic model checking *[4] algorithms, letting us construct attack graphs automatically and efficiently. We also describe two analyses to help decide which attacks would be most cost effective to guard against. We implemented our technique in a tool suite and tested it on a small network example, which includes models of a firewall and an intrusion detection system.*

1. Overview

As networks of hosts continue to grow in size and complexity, evaluating their vulnerability to attack becomes increasingly more important to automate. There are several tools, such as COPS [10] and Renaud Deraison's Nessus Security Scanner [9], that report vulnerabilities of individual hosts. To evaluate the vulnerability of a network of hosts, however, we also have to analyze the effects of interactions of local vulnerabilities and find global vulnerabilities introduced by the interconnections between hosts. A typical process for vulnerability analysis of a network proceeds as follows. First, we determine vulnerabilities of individual hosts using scanning tools, such as COPS and Nessus Scanner. Using this local vulnerability information along with other information about the network, such as connectivity between hosts, we then produce *attack graphs*. Each path in an attack graph is a series of exploits, which we call *atomic attacks*, that leads to an undesirable state, e.g., a state where an intruder has obtained administrative access to a critical host. We can then perform further analyses, such as risk analysis [21], reliability analysis [13], or shortest path analysis [23], on the attack graph to assess the overall vulnerability of the network.

Constructing attack graphs is a crucial part of doing vulnerability analysis of a network of hosts. Construction by hand, however, is tedious, error-prone, and impractical for attack graphs larger than a hundred nodes. Automating the process of constructing attack graphs also ensures that the attack graphs are *exhaustive* and *succinct*. An attack graph is exhaustive if it covers all possible attacks, and succinct if it contains only those network states from which the intruder can reach his goal. We follow these steps to produce and analyze attack graphs:

1. **Model the network.**

We model the network as a finite state machine, where state transitions correspond to atomic attacks launched by the intruder. We also specify a desired security property (e.g., an intruder should never obtain root access to host A). The intruder's goal generally corresponds to violating this property.

2. **Produce an attack graph.**

Using the model from Step 1, our modified version of the model checker NuSMV [16] automatically produces the attack graph. The graphs are rendered using the GraphViz visualization package [1].

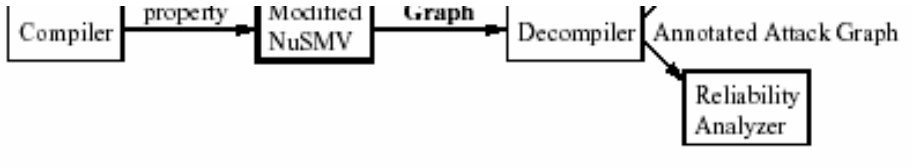


Figure 1. Tool Suite

3. **Analysis of attack graphs.**

A raw attack graph is a low-level state transition diagram. To allow the domain specialist to analyze it in a meaningful way, we parse the graph and reconstruct the original meanings of the state variables as they relate to the network intrusion domain. In Section 4 we discuss two different analyses on attack graphs that quantify the likelihood of intruder success.

Figure 1 shows the architecture of our tool suite. We do not require or expect users of our tool suite to have model checking expertise. Instead of using the input language of the NuSMV model checker, a user may describe the network model and desired property in XML [5]. We built a special-purpose compiler that takes an XML description and translates it into the input language of NuSMV.

In the field of model checking, the use of fundamental data structures, such as *Binary Decision Diagrams* (BDDs) [2], enabled significant advances in the size of the systems that can be analyzed [3, 4]. More recently, model checking researchers have developed a variety of reduction and abstraction techniques to handle even larger, possibly infinite state spaces. Since our techniques build upon the underlying representation and algorithms used in model checking, we are able to leverage the recent success in that field. As model checkers handle larger state spaces, our analysis can be applied to larger networks.

Our paper reports on the following contributions to analyzing vulnerabilities in networks:

- We exhibit an algorithm for automatic generation of attack graphs. The algorithm generates exhaustive and succinct attack graphs. We provide a tool, as a part of a larger tool suite, which implements the algorithm.
- Through a small case study, we identify a level of atomicity appropriate for describing a model of the network and an intruder's arsenal of atomic attacks. The model is abstract enough to be understood by security domain experts, yet simple enough for our tool to analyze efficiently.
- Our network model includes intrusion detection components and distinguishes between stealthy and detectable attack variants. We are able to generate "stealthy" attack subgraphs (i.e. subgraphs with attacks that are not detected by the intrusion detection components). Analysis of stealthy attack subgraphs reveals the best locations for placing additional intrusion detection components.
- We describe two ways of analyzing attack graphs: an algorithm for determining a minimal set of atomic attacks whose prevention would guarantee that the intruder will fail, and a probabilistic reliability analysis that determines the likelihood that the intruder will succeed.

Paper organization: We give a detailed description of our attack graph generation algorithm in Section 2. We describe an intrusion detection case study in Section 3 and results of attack graph analysis in Section 4. We discuss related work in Section 5 and close with suggestions for future work in Section 6.

2. Attack Graphs

First, we define formally *attack graphs*, the data structure used to represent all possible attacks on a network.

Definition 1 An *attack graph* or *AG* is a tuple $G = (S, \tau, S_0, S_s)$, where S is a set of states, $\tau \subseteq S \times S$ is a transition relation, $S_0 \subseteq S$ is a set of initial states, and $S_s \subseteq S$ is a set of success states.

Intuitively, S_s denotes the set of states where the intruder has achieved his goals. Unless stated otherwise, we assume that the transition relation τ is total. We define an *execution fragment* as a finite sequence of states $S_0S_1...S_n$ such that $(S_i, S_{i+1}) \in \tau$ for all $0 \leq i \leq n$. An execution fragment with $S_0 \in S_0$ is an *execution*, and an execution whose final state is in S_0 is an *attack*, i.e., the execution corresponds to a sequence of atomic attacks leading to the intruder's goal state.

2.1. Constructing Attack Graphs

Model checking is a technique for checking whether a formal model M of a system satisfies a given property ρ. If the property is false in the model, model checkers typically output a counter-example, or a sequence of transitions ending with a violation of the property.

In the model checker NuSMV, the model *M* is a finite labeled transition system and ρ is a property written in *Computation Tree Logic* (*CTL*). In this paper, we consider only safety properties, which in CTL have the form $\mathbf{AG}f$ (i.e., $\rho = \mathbf{AG}f$, where f is a formula in propositional logic). If the model *M* satisfies the property ρ, NuSMV reports "true." If *M* does not satisfy ρ, NuSMV produces a counterexample. In our context *M* is a model of the network and ρ is a safety property. A single counter-example shows an attack that leads to a violation of the safety property.

Attack graphs depict ways in which an intruder can force a network into an unsafe state. We can express the property that an unsafe state cannot be reached as:

$$\mathbf{AG}(\neg unsafe)$$

When this property is false, there are unsafe states that are reachable from the initial state. The precise meaning of *unsafe* depends on the network. For example, the property given below might be used to say that the privilege level of the adversary on the host with index 2 should always be less than the root (administrative) privilege.

Input:

$\boldsymbol{S}$ – set of states
$\boldsymbol{R} \subseteq \boldsymbol{S} \times \boldsymbol{S}$ – transition relation
$\boldsymbol{S_0} \subseteq \boldsymbol{S}$ – set of initial states
$\boldsymbol{L : S \rightarrow 2^{AP}}$ - labeling of states with propositional formulas
$\boldsymbol{P} = \mathbf{AG}(\neg unsafe)$(a safety property)

Output:

attack graph $G_p = (S_{unsafe}, R^p, S^p_0, S^p_s)$

Algorithm: *Generate Attack Graph (S, R, S_0, L, p)*

(* Use model checking to find the set of states S_{unsafe} that violate the safety property $\mathbf{AG}(\neg unsafe)$. *)
$S_{unsafe} = modelCheck(S, R, S_0, L, p)$
(* Restrict the transition relation *R* to states in the set S_{unsafe} *)
$R^p = R \cap (S_{unsafe} \times S_{unsafe})$.
$S^p_0 = S_0 \cap S_{unsafe}$
$S^p_s = \{s \mid s \in S_{unsafe} \wedge unsafe\ L(s)\}$.
$Return(S_{unsafe}, R^p, S^p_0, S^p_s)$.

Figure 2. Algorithm for Generating Attack Graphs

AG(*network.adversary.privilege [2] < network.priv.root*)

We briefly describe the algorithm for constructing attack graphs for the property ***AG(¬unsafe)***. The first step is to determine the set of states S_τ that are reachable from the initial state. Next, the algorithm computes the set of reachable states S_{unsafe} that have a path to an unsafe state. The set of states S_{unsafe} is computed using an iterative algorithm derived from a fix-point characterization of the ***AG*** operator [4]. Let R be the transition relation of the model, i.e., $(s, s') \in R$ if and only if there is a transition from state s to s'. By restricting the domain and range of R to S_{unsafe} we obtain a transition relation R_p that encapsulates the edges of the attack graph. Therefore, the attack graph is $(S_{unsafe}, R^p, S_0^p, S_s^p)$, where S_{unsafe} and R^p represent the set of nodes and set of edges of the graph, respectively; $S_0^p = S_0 \cap S_{unsafe}$ is the set of initial states; and $S_s^p = \{s \mid s \in S_{unsafe} \wedge unsafe \in L(s)\}$ is the set of success states. This algorithm is given in Figure 2.

In symbolic model checkers, such as NuSMV, the transition relation and sets of states are represented using BDDs [2], a compact representation for boolean functions. There are efficient BDD algorithms for all operations used in our algorithm.

2.2. Attack Graph Properties

We can show that an attack graph *G* generated by the algorithm in Figure 2 is exhaustive (Lemma 1a) and succinct (Lemma 1b). Whereas succinctness is a property about states in an attack graph, Lemma 1c states a similar property for transitions. Appendix A contains a proof of the lemma.

Lemma 1

(a) (Exhaustive) An execution *e* of the input model (S, R, S_0, L) violates the property $\rho = \mathbf{AG}(\neg unsafe)$ if and only if *e* is an attack in the attack graph $G = (S_{unsafe}, R^p, S_0^p, S_s^p)$.

(b) (Succinct states) A state *s* of the input model (S, R, S_0, L) is in the attack graph G if and only if there is an attack in G that contains *s*.

(c) (Succinct transitions) A transition $t = (s_1, s_2)$ of the input model (S, R, S_0, L) is in the attack graph G if and only if there is an attack in G that includes *t*.

3. An Intrusion Detection Example

Consider the example network shown in Figure 3. There are two target hosts, ip_1 and ip_2, and a firewall separating them from the rest of the Internet. As shown, each host is running two of three possible services (ftp, sshd, a database). An intrusion detection system (IDS) watches the network traffic between the target hosts and the outside world. There are four possible atomic attacks, identified numerically as follows: (0) sshd buffer overflow, (1) ftp .rhosts, (2) remote login, and (3) local buffer overflow (an explanation of each attack follows). If an atomic attack is *detectable*, the intrusion detection system will trigger an alarm; if an attack is *stealthy*, the IDS misses it. The ftp .rhosts attack needs to find the target host with two vulnerabilities: a writable home directory and an executable command shell assigned to the ftp user name. The local buffer overflow exploits a vulnerable version of the xterm executable.

The intruder launches his attack starting from a single computer, ip_a, which lies outside the firewall. His eventual goal is to disrupt the functioning of the database. For that, the intruder needs root access on the database host ip_2.

We construct a finite state model of the network so that each state transition corresponds to a single atomic attack by the intruder. A state in the model represents the state of the system between atomic attacks. A typical transition from state s_1 to state s_2 corresponds to an atomic attack whose preconditions are satisfied in s_1 and whose postconditions hold in state s_2. An *attack* is a sequence of state transitions culminating in the intruder achieving his goal. The entire attack graph is thus a representation of all the ways the intruder can succeed.

3.1. Finite State Model

The network. We model a network as a set of facts, each represented as a relational predicate. The state of the network specifies services, host vulnerabilities, connectivity between hosts, and a remote login trust relation. Following Ritchey and Ammann [20], connectivity is expressed as a ternary relation $R \subseteq Host \times Host \times Port$, where $R(h_1, h_2, p)$ means that host h_2 is reachable from host h_1 on port p. Note that the connectivity relation incorporates firewalls and other elements that restrict the ability of one host to connect to another. Slightly abusing notation, we say $R(h_1, h_2)$ when there is a network route from h_1 to h_2. Similarly, we model trust as a binary relation $T_R \subseteq Host \times Host$, where $T_R(h_1, h_2)$ indicates that a user may log in from host h_2 to host h_1 without authentication (i.e., host h_1 "trusts" host h_2).

Initially, there is no trust between any of the hosts; the trust relation T_R is empty. The connectivity relation R is shown in the following table. An entry in the table corresponds to a pair of hosts (h_1, h_2). Each entry is a triple of boolean values. The first value is 'y' if h_1 and h_2 are connected by a physical link, the second value is 'y' if h_1 can connect to h_2 on the ftp port, and the third value is 'y' if h_1 can connect to h_2 on the sshd port.

The intruder. The intruder has a store of knowledge about the target network and its users. This knowledge includes host addresses, known vulnerabilities, information about running services, etc. The function $plvl_A$*: Hosts → {none, user, root}* gives the level of privilege that intruder A has on each host. There is a total order on the privilege levels: *none* < *user* < *root*. Initially, the intruder has root access on his own machine ip_a, but no access to the other hosts.

R	ip_a	ip_1	ip_2
ip_a	y,n,n	y,y,y	y,y,n
ip_1	y,n,n	y,y,y	y,y,n
ip_2	y,n,n	y,y,y	y,y,n

Intrusion detection system. Atomic attacks are classified as being either *detectable* or *stealthy* with respect to the Intrusion Detection System (IDS). If an attack is detectable, it will trigger an alarm when executed on a host or network segment monitored by the IDS. If an attack is *stealthy*, the IDS does not see it.

We specify the IDS with a function *ids: Host* x *Host* x *Attack* → {*d, s, b*}, where $ids(h_1, h_2, a) = d$ if attack a is *d*etectable when executed with source host h_1 and target host h_2; $ids(h_1, h_2, a) = s$ if attack a is *s*tealthy when executed with source host h_1 and target host h_2; and $ids(h_1, h_2, a) = b$ if attack a has *b*oth detectable and stealthy strains, and success in detecting the attack depends on which strain is used. When h_1 and h_2 refer to the same host, $ids(h_1, h_2, a)$ specifies the intrusion detection system component (if any) located on that host. When h_1 and h_2 refer to different hosts, $ids(h_1, h_2, a)$ specifies the intrusion detection system component (if any) monitoring the network path between h_1 and h_2.

In addition, a global boolean variable specifies whether the IDS alarm has been triggered by any previously executed atomic attack.

In our example, the paths between (ip_a, ip_1) and between (ip_a, ip_2) are monitored by a single network-based IDS. The path between (ip_1, ip_2) is not monitored. There are no host-based intrusion detection components.

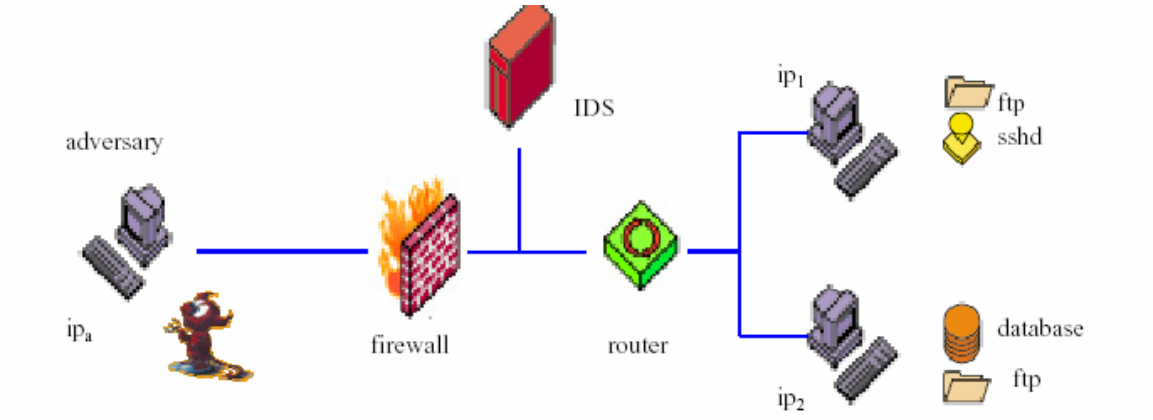


Figure 3. Example Network

Atomic Attacks. We model four atomic attacks:

1. *sshd buffer overflow*: This remote-to-root attack immediately gives a remote user a root shell on the target machine. It has detectable and stealthy variants.

2. *ftp .rhosts*: Using an ftp vulnerability, the intruder creates an .rhosts file in the ftp home directory, creating a remote login trust relationship between his machine and the target machine. This attack is stealthy.

3. *remote login*: Using an existing remote login trust relationship between two machines, the intruder logs in from one machine to another, getting a user shell without supplying a password. This operation is usually a legitimate action performed by regular users, but from the intruder's point of view, it is an atomic attack. This attack is detectable.

4. *local buffer overflow*: If the intruder has acquired a user shell on the target machine, the next step is to exploit a buffer overflow vulnerability on a setuid root file to gain root access. The intruder may transfer the necessary binary code via ftp (or scp) or create it locally using an editor such as vi. This attack is stealthy.

Each atomic attack is a rule that describes how the intruder can change the network or add to his knowledge about it. A specification of an atomic attack has four components: *intruder preconditions*, *network preconditions*, *intruder effects*, and *network effects*. The *intruder preconditions* component lists the intruder's capabilities and knowledge required to launch the atomic attack. The *network preconditions* component lists the facts about the network that must hold before launching the atomic attack. Finally, the *intruder* and *network effects* components list the attack's effects on the intruder and on the network state, respectively. For example, the sshd buffer overflow attack is specified as follows:

```
attack sshd-buffer-overflow is
    intruder preconditions
        (* User-level privileges on host S *)
        plvl_A(S) ≥ user
        No root-level privileges on host T *)
        plvl_A(T) < root
    network preconditions
        (* Host T is running sshd *)
        ssh_T
        (* Host T is reachable from S on port sp *)
        R(S, T, sp)
    intruder effects
        (* Root-level privileges on host T *)
        plvl_A(T) := root
    network effects
        (* Host T is not running sshd *)
        ¬ssh_T
end
```

3.2. NuSMV Encoding

It is necessary to ensure that the model checker considers all atomic attacks in each state, so that the resulting attack graph enumerates all possible attacks. So the model checker must choose attacks nondeterministically, subject to preconditions being fulfilled. We also allow nondeterministic choices for the source host and the target host of each atomic attack. The NuSMV encoding of the model contains nondeterministically assigned state variables that specify:

- which attack (concretely, an attack number) will be tried next
- the *source* host from which the atomic attack will be initiated
- the *target* host of the atomic attack
- whether the next attack is detectable or stealthy with respect to a given intrusion detection system. This variable is set deterministically when the next attack is known to be detectable or stealthy. When the next attack has both detectable and stealthy strains, the variable is set nondeterministically.

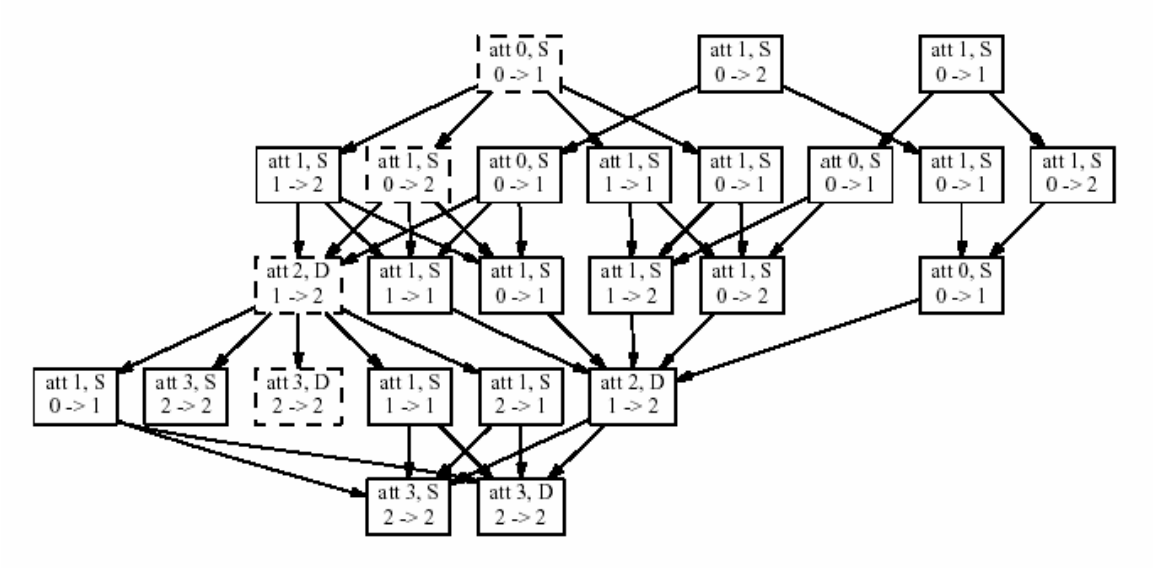


Figure 4. Attack Graph

In an effort to reduce the state space of the model, the NuSMV encoding restricts the legal states to those where the attack number, source, and target variables correspond to an enabled attack. In addition, when a variable's value is irrelevant in a particular context, we deterministically set the variable to a fixed value in that context. As an example, when the next attack is local to one host, we force the value of the variable designating the source host of the attack to be the same as the target host of the attack.

3.3. Experimental Results: Attack Graphs

Recall that the goal of our intruder is to obtain access to the database service running on host ip_2. For that, the intruder needs to get root access on ip_2 without triggering an IDS alarm. Thus, the property we want to violate (in order to get the attack graph) is that either an intruder never gets root privilege on host ip_2 or he is detected by the IDS:

AG(*network.adversary.privilege[2] < network.priv.root* | *network.detected)*

Figure 4 shows the attack graph produced by NuSMV for this property. Each node is labeled by an attack id number (see table below), which corresponds to the atomic attack *to be attempted next*; a flag S/D indicating whether the attack is stealthy or detectable by the intrusion detection system; and the numbers of the source and target hosts. The following tables show attack and host numbers.

no.	attack
0	sshd buffer overflow
1	ftp .rhosts
2	remote login
3	local buffer overflow

no.	host
0	ip_a
1	ip_1
2	ip_2

Any path in the graph from a root node to a leaf node shows a sequence of atomic attacks that the intruder can employ to achieve his goal while remaining undetected. For instance, the path highlighted by double boxed nodes consists of the following sequence of four atomic attacks: overflow sshd buffer on host 1, overwrite .rhosts file on host 2 to establish rsh trust between hosts 1 and 2, log in using rsh from host 1 to host 2, and finally, overflow a local buffer on host 2 to obtain root privileges.

3.4. Performance Observations

We conducted the experiments on a Pentium III/1Ghz RAM running RedHat Linux 7.0.

The NuSMV encoding of the simple network in Figure 3 has 91 bits of state (i.e., potentially 2^{91} states), but only 101 states are reachable. The tool automatically found an appropriate BDD variable ordering under which the run time of the tool on this example is about 5 seconds.

To gauge how the run time depends on the scale of the model, we enlarged the example with two additional hosts, four additional atomic attacks, several new vulnerabilities, and flexible firewall configurations. The enlarged model has 229 bits of state and 6190 reachable states. The attack graph has 5948 nodes and 68364 edges. NuSMV took 2 hours to construct the attack graph for this model; however, the model checking part took only 5 minutes. The performance bottleneck is inside our graph generation procedure, and we are working on performance enhancements.

4. Analysis of Attack Graphs

Once we have an attack graph generated for a specific network with respect to a given safety property, the user may wish to probe it for further analysis. For example, an analyst may be faced with a choice of deploying either additional network attack detection tools or prevention techniques. Which would be more cost-effective to deploy? In doing the minimization analysis described in Sections 4.1 through 4.3, the analyst can determine a minimal set of atomic attacks that must be prevented to guarantee that the intruder cannot achieve his goal. In doing the reliability analysis described in Section 4.4, the analyst can determine the likelihood that an intruder will succeed or the likelihood that the IDS will detect his attack activity.

4.1. Minimization Analysis

Given a fixed set of atomic attacks, not all of them may be available to the intruder. Can we find a minimal set of atomic attacks that we should prevent so that the intruder fails to achieve his goal? To answer this question, we modify the model slightly, making only a subset of atomic attacks available to the intruder. For simplicity, we nondeterministically decide which subset to consider initially, before any attack begins; once the choice is made, the subset of available atomic attacks remains constant during any given attack. We ran the model checker on the modified model with the invariant property that says the intruder never gets root privilege on host ip_2:

AG(*network.adversary.privilege[2]* < *network.priv.root*)

The post-processor marked the states where the intruder has been detected by the IDS. The result is shown in Figure 5. The white rectangles indicate states where the attacker had not yet been detected by the intrusion detection system. The black rectangles are states where the intrusion detection system has sounded the alarm. Thus, white leaf nodes are desirable for the attacker in that the objective is achieved without detection. Black leaf nodes are less desirable, the attacker achieves his objective, but the alarm goes off.

The resolution of which atomic attacks are available to the intruder happens in the circular nodes near the root of the graph. The first transition out of the root (initial) state picks the subset of attacks that the intruder will use. Each child of the root node is itself the root of a disjoint subgraph where the subset of atomic attacks chosen for that child is used. Note that the number of such subgraphs descending from the root node corresponds to the number of subsets of atomic attacks with which the intruder can be successful. The model checker determines that for any other possible subset, there is no possible successful sequence of atomic attacks.

The root of the graph in Figure 5 has two subgraphs, corresponding to two subsets of atomic attacks that will allow the intruder to succeed. In the left subgraph the sshd buffer overflow attack is not available to the intruder; it can readily be seen that the intruder can still succeed, but cannot do so while

remaining undetected by the IDS. In the right subgraph, all attacks are available. Thus, the entire attack graph implies that all atomic attacks other than the sshd attack are indispensable: the intruder cannot succeed without them. The analyst can use this information to guide decisions on which network defenses can be profitably upgraded.

The white cluster in the middle of the figure is isomorphic to the scenario graph presented in Figure 4; it shows the ways in which the intruder can achieve his objective without detection (i.e., all paths by which the intruder reaches a white leaf in the graph).

Checking every possible subset of attacks is exponentialing the number of attacks. In the next subsection, we show that finding the *minimum* set of atomic attacks which must be removed to thwart the intruder is in fact *NP*-complete. Then in the following subsection we also show how a *minimal* set can be found in polynomial-time.

4.2. Minimum and Minimal Critical Attack Sets

Assume that we have produced an attack graph corresponding to the following safety property:

$$\boldsymbol{AG}(\neg unsafe)$$

Let A be the set of attacks. Let $G = (S, E, s_0, L)$ be the attack graph, where S is the set of states, $E \subseteq S \times S$ is the set of edges, $s_0 \in S$ is the initial state, and $L: E \rightarrow A \cup \{\varepsilon\}$ is a labeling function where $L(e) = a$ if an edge $e = (s \rightarrow s')$ corresponds to an attack a, otherwise $L(e) = e$. Given a state $s \in S$, a set of attacks C is *critical* with respect to s if and only if the intruder cannot reach his goal from s when the attacks in C are removed from his arsenal A. Equivalently, C is critical with respect to s if and only if every path from s to an unsafe state has at least one edge labeled with an attack $a \in C$.

A critical set corresponding to a state s is *minimal* (denoted $A(s)$) if no subset of $A(s)$ is critical with respect to s. A critical set corresponding to a state s is *minimum* (denoted $M(s)$) if there is no critical set $M'(s_0)$ such that $|M'(s)| < |M(s)|$. In general, there can be multiple minimum and multiple minimal critical sets corresponding to a state s. Of course, all minimum critical sets must be of the same size.

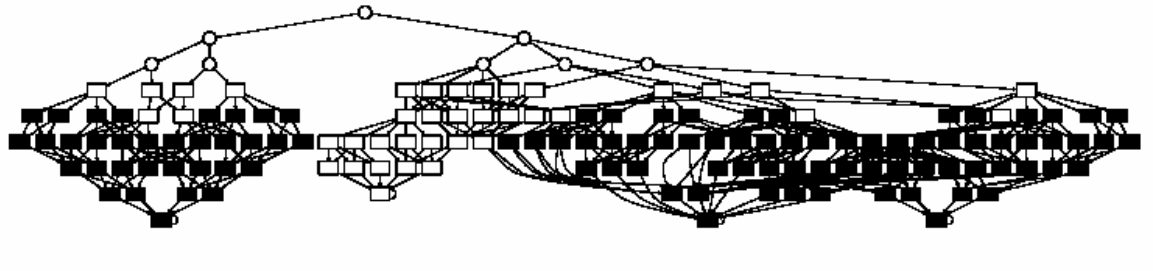


Figure 5. Attack Graph Analysis

Given an attack graph $G = (S, E s_0, L)$, consider the problem of finding a minimum critical set of attacks $M(s_0)$. We will call this problem the Minimum Critical Set of Attacks (MCSA) problem. We prove that the decision version of MCSA is *NP*-complete.

Lemma 2 Assume that we are given an attack graph $G = (S, E s_0, L)$, and an integer k. The problem of determining whether there is a critical set $C(s_0)$ such that $|C(s_0)| \leq k$ is *NP*-complete.

Proof Sketch: First, we prove that the problem is in *NP*. Guess a set $C \subseteq A$ with size $\leq$ k. We need to check that C is a critical set of attacks. This can be accomplished in polynomial time using the procedure *isCritical(G, C)* described below. Therefore, the problem is in *NP*.

To prove that the problem is *NP*-hard, we give a reduction from the *minimum cover* problem [11, Page 222]. See Appendix B for the remaining details of the proof.

4.3. Computing Minimal Critical Sets

Consider now the problem of finding a minimal critical set $A(s_0)$ corresponding to the initial state s_0. We give an algorithm for computing $A(s_0)$ that runs in time $O(mn)$, where $m = |S| + |E|$ is the size of the attack graph G and $n = |A|$ is the number of attacks. First, we describe a procedure *isCritical (G, C),* which determines whether a set $C \subseteq A$ is a critical set corresponding to the initial state s_0. This procedure runs in $O(m)$ time. We simply delete all edges from G that are labeled with an action from the set C. After that, if an unsafe state is still reachable from the initial state s_0, then C is not a critical set (because there is a path from s_0 to an unsafe state which does not use an attack from the set C). This step can be performed in $O(m)$ time using standard graph algorithms [6]. The algorithm starts with A as the empty set. At each step of the algorithm we perform the following procedure:

if *isCritical (G, C)* returns true, the algorithm stops and returns A. Otherwise, pick an $a \in a \backslash C$ and add it to the set C.

We start with an empty set and keep adding attacks until we obtain a critical set. Notice that since A is a critical set, the number of steps taken by the algorithm is at most n. Each step takes $O(m)$ time, so that the worst case running time of the algorithm is $O(mn)$. If attacks have costs associated with them, then at each step we can pick an attack that has the minimum cost, i.e., pick an $a \in A \backslash C$ with the minimum cost. This will bias the procedure to pick sets with lower cost.

Next, we show how the procedure described above can be carried out using model checking. Assume that the set of attacks is A is $(a_1, \ldots, a_n)$. We associate a boolean variable x_i with each attack a_i. If attack a_i is activated (the intruder can use the attack), $x_{i=1}$, otherwise $x_i = 0$. The variable x_i appears in the precondition corresponding to the attack a_i. Initially, all x_i s are set to 0, representing that the set C is empty. Notice that if the model checker returns a counter-example, then there is a path from the initial state to an unsafe state. Recall that the specification is:

$$\mathbf{AG}(\neg unsafe)$$

Now in each step in the procedure, we pick an index i such that $x_i = 0$ and set $x_i = 0$. We stop the first time the model checker provides a counter-example. The set of attacks whose corresponding variables are set to 1 represents a critical set. The worst case complexity of this procedure is the same as the one given before, but in practice symbolic model checkers, such as NuSMV, will perform efficiently. Intuitively, we are using the model checker to implement the procedure *isCritical(G,C)*.

4.4. Probabilistic Reliability Analysis

When empirical information about the likelihood of certain events in the network is available, we can use well known graph algorithms to answer quantitative questions about the attack graph. Suppose we know the probabilities of some transitions in the scenario graph. After appropriately annotating the attack graph with these probabilities, we can interpret it as a Markov Decision Process (see [12] for details).

The standard MDP *value iteration algorithm* [19] computes the optimal policy for selecting actions in an MDP that results in maximum benefit (or minimum cost) for the decision maker. Value iteration can compute the worst case probability of intruder success in an attack graph as follows. We assign all nodes where the intruder's goal has been achieved the benefit value of 1, and all other nodes the benefit value of 0. Then we run the value iteration algorithm. The algorithm finds the optimal attack selection policy for the intruder and assigns the expected benefit value resulting from that policy to each state in the scenario graph. The expected value is a fraction of 1, and it is equivalent to the probability of getting to the goal state from that node, assuming the intruder always follows the optimal policy.

We implemented the value iteration algorithm in an attack graph post-processor ("Reliability Analyzer" of Figure 1) and ran it on a slightly modified version of our example. In the modified example

each attack has both detectable and stealthy variants. We assumed that for a typical network, a certain percentage of attempted intrusions is performed by sophisticated attackers who keep on top of latest IDS technology and use stealthy attack variants. We arbitrarily assigned probabilities of detecting each atomic attack as follows: 0.8 for sshd buffer overflow, 0.5 for ftp .rhosts, 0.95 for the remote login, and 0.2 for local buffer overflow. The intruder's goal is to get root access at host ip_2 while remaining undetected. Accordingly, the states where this goal has been achieved were assigned benefit value 1.

In this setup, the computed probability of intruder success is 0.2, and his best strategy is to attempt sshd buffer overflow on host ip_1, and then conduct the rest of the attack from that host. The only possibility of detection is the sshd buffer overflow attack itself, since the IDS does not see the activity between hosts ip_1 and ip_2.

The system administrator can use this technique to evaluate effectiveness of various security fixes. For instance, installing an additional IDS component to monitor the network traffic between hosts ip_1 and ip_2 reduces the probability of the intruder remaining undetected to 0.025; installing a host-based IDS on host ip_2 reduces the probability to 0.16. Other things being equal, this is an indication that the former remedy is more effective.

5. Related Work

The work by Phillips and Swiler [18] is the closest to ours. They propose the concept of attack graphs that is similar to the one described here. However, they take an "attack-centric" view of the system. Since we work with a general modeling language, we can express in our model both seemingly benign system events (such as failure of a link) and malicious events (such as attacks). Therefore, our attack graphs are more general than the one proposed by Phillips and Swiler. Recently, Swiler *et al.* describe a tool [23] for generating attack graphs based on their previous work. Their tool constructs the attack graph by forward exploration starting from the initial state. A symbolic model checker (like NuSMV) works backward from the goal state to construct the attack graph. A major advantage of the backward algorithm is that vulnerabilities that are not relevant to the safety property (or the goal of the intruder) are never explored. Our approach can result in significant savings in space. (Swiler *et al.* refer to the advantages of the backward search in their paper [23].) More generally, the advantage of using model checking instead of forward search is that the technique can be expanded to include liveness properties, which can model service guarantees in the face of malicious activity.

Moreover, by using model checking we leverage all the advanced techniques developed in that area. For example, the *cone of influence reduction* [14] in model checking abstracts away part of the system that is not relevant to the specification. In our context, if there is a vulnerability that is not relevant to a safety property, it will not be considered during model checking. Finally, the attack graph analysis suggested by Phillips and Swiler is different from the ones presented in this paper. We plan to incorporate their analysis into our tool suite.

Templeton and Levitt [24] propose a requires/provides model for attacks. The model links atomic attacks into scenarios, with earlier atomic attacks supplying the prerequisites for the later ones. Templeton and Levitt point out that relating seemingly innocuous system behavior to known attack scenarios can help discover new atomic attacks. However, they do not consider combining their attack scenarios into attack graphs.

Dacier [8] proposes the concept of privilege graphs. Each node in the privilege graph represents a set of privileges owned by the user; edges represent vulnerabilities. Privilege graphs are then explored to construct attack state graphs, which represents different ways in which an intruder can reach a certain goal, such as root access on a host. He also defines a metric, called the *mean effort to failure* or METF, based on the attack state graphs. Orlato et al. describe an experimental evaluation of a framework based on these ideas [17]. At the surface, our notion of attack graphs seems similar to the one proposed by Dacier. However, as is the case with Phillips and Swiler, Dacier takes an "attack-centric" view of the world. As pointed out above, our attack graphs are more general. From the experiments conducted by Orlato *et al.* it appears that even for small examples the space required to construct attack state graphs becomes

prohibitive. By basing our algorithm on model checking we take advantage of advances in representing large state spaces and can thus hope to represent large attack graphs. We can perform the analytical analysis proposed by Dacier on attack graphs constructed by our tool. We also plan to conduct an experimental evaluation similar to the one performed by Orlato et al.

Ritchey and Ammann [20] also use model checking for vulnerability analysis of networks. They use the (unmodified) model checker SMV [22]. They can obtain only one counter-example, i.e., only one attack corresponding to an unsafe state. In contrast, we modified the model checker NuSMV to produce attack graphs, representing all possible attacks. We also described post-facto analyses that can be performed on these attack graphs. These analysis techniques cannot be meaningfully performed on single attacks.

Graph-based data structures have also been used in network intrusion detection systems, such as *NetSTAT* [25]. There are two major components in NetSTAT, a set of probes placed at different points in the network and an analyzer. The analyzer processes events generated by the probes and generates alarms by consulting a network fact base and a scenario database. The network fact base contains information (such as connectivity) about the network being monitored. The scenario database has a directed graph representation of various atomic attacks. For example, the graph corresponding to an IP spoofing attack shows various steps that an intruder takes to mount that specific attack. The authors state that "in the analysis process the most critical operation is the generation of all possible instances of an attack scenario with respect to a given target network." Therefore, we believe that our tool can help network intrusion detection systems, such as NetSTAT, in automatically producing attack scenarios. We leave this as a future direction for research.

Cuppens and Ortalo [7] propose a declarative language (LAMBDA) for specifying attacks in terms of pre- and postconditions. LAMBDA is a superset of the simple language we used to model attacks in our work. The language is modular and hierarchical; higher-level attacks can be described using lower-level attacks as components. LAMBDA also includes intrusion detection elements. Attack specifications include information about the steps needed to detect the attack and the steps needed to verify that the attack has already been carried out. We are studying the possibility converting our representation of attacks to LAMBDA.

6. Future Work

We have so far restricted our work to only safety (invariant) properties. To exploit the full power of model checking, we need a method of generating attack graphs for more general classes of properties. For example, the following liveness property states that a user will always be able to access a server whenever he wants to.

$$\mathbf{AG}(server.user.request \rightarrow \mathbf{AF}(server.user.access))$$

This property would not be true if the server can be disabled using a denial-of-service attack. We plan to explore generation of attack graphs for universally quantified fragments of Computational Tree Logic and Linear Temporal Logic.

To make our tool suite more usable by security experts and system administrators, we see the value of building a library of specifications of atomic attacks. Our hope is that increasing this arsenal of specifications outpaces the growth in the arsenal of known attacks. Furthermore, one reason model checking has been so successful is that it discovers unknown bugs in hardware circuits and protocols 1. Analogously, by using our tool suite based on the power of model checking techniques, we can potentially discover new, unexpected attacks, and hence identify new network vulnerabilities.

In principle, our technique is not limited to modeling attacks only. The expressive power of model checkers lets us model benign system activity as well. We believe that the ability of modern model checkers to handle more complex properties can be adapted to our tool. For example, "liveness" properties such as "a legitimate user's transaction will finish despite intruder interference" are easily specified in

temporal logic and checked by a model checker. Unlike invariants, such properties cannot be handled by simple Reachability analysis or other classical graph algorithms. Adapting the power of model checking to analyze such properties opens a promising research direction in automated security analysis.

References

[1] AT & T Labs. *Graphviz - open source graph drawing software*. *http://www.research.att.com/sw/tools/graphviz/*.

[2] R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677-691, Aug. 1986.

[3] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[5] W. Consortium. Extensible Markup Language (XML) 1.0. *http://www.w3.org/TR/REC-xml*, February 1998.

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1985.

[7] F. Cuppens and R. Ortalo. Lambda: A language to model a database for detection of attacks. In *Proceedings of the Third International Workshop on the Recent Advances in Intrusion Detection (RAID)*, number 1907 in LNCS, pages 197–216. Springer-Verlag, 2000.

[8] M. Dacier. *Towards Quantitative Evaluation of Computer Security*. PhD thesis, Institut National Polytechnique de Toulouse, Dec. 1994.

[9] R. Deraison. Nessus Scanner. *http://www.nessus.org*.

[10] D. Farmer and E. Spafford. The COPS security checker system. In *Proceedings of the Summer Usenix Conference*, 1990.

[11] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, 1979.

[12] S. Jha, O. Sheyner, and J. M. Wing. Minimization and reliability analyses of attack graphs. Technical Report CMUCS- 02-109, Carnegie Mellon University, February 2002.

[13] S. Jha and J.Wing. Survivability analysis of networked systems. In *Proceedings of the International Conference on Software Engineering*, Toronto, Canada, May 2001.

[14] R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[15] K. McMillan and J. Schwalbe. Formal verification of the Gigamax cache consistency protocol. In N. Suzuki, editor, *Shared Memory Multiprocessing*. MIT Press, 1992.

[16] NuSMV. NuSMV: A New Symbolic Model Checker. http://afrodite.itc.it:1024/~nusmv/.

[17] R. Ortalo, Y. Dewarte, and M. Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25(5):633–650, September/October 1999.

[18] C. Phillips and L. Swiler. A graph-based system for network vulnerability analysis. In *New Security Paradigms Workshop*, pages 71–79, 1998.

[19] M. Puterman. *Markov Decision Processes*. John Wiley & Sons, New York, NY, 1994.

[20] R. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–165, May 2001.

[21] B. Schneier. Modeling security threats. *Dr. Dobb's Journal*, December 1999.

[22] SMV. SMV: A Symbolic Model Checker. http:// www.cs.cmu.edu/~modelcheck/.

[23] L. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. *In Proceeding of the DARPA Information Survivability Conference and Exposition*, June 2000.

[24] S. Templeton and K. Levitt. A requires/provides model for computer attacks. In *Proceedings of the New Security Paradigms Workshop*, Cork, Ireland, 2000.

[25] G. Vigna and R. Kemmerer. Netstat: A network-based intrusion detection system. *Journal of Computer Security*, 7(1), 1999.

A. Exhaustive and Succinct Attack Graphs

Lemma 1: (a) (Exhaustive) An execution e of the input model (S, R, S_0, L) violates the property p = ***AG(¬unsafe)*** if and only if e is an attack in the attack graph G = *(S¬unsafe, Rp, Sp0, Sps)*.
(b) (Succinct states) A state s of the input model (S, R, S_0, L) is in the attack graph G if and only if there is an attack in G that contains s.
(c) (Succinct transitions) A transition $t = (s_1, s_2)$ of the input model (S, R, S_0, L) is in the attack graph G if and only if there is an attack in G that includes t.

Proof:

(a) (=>) Let $e = s_0 t_0 \ldots t_{n-1} s_n$ be a (finite) execution of the input model such that s_n is an *unsafe* state. To prove that e is an attack in G, it is sufficient to show (1) $s_0 \in S_0^p$, (2) $s_n \in S_s^p$, and (3) for all $0 \leq k \leq n$, $s_k \in S$ and $t_k \in R^p$.

Since *unsafe* holds at s_n and for all k there is a path from s_k to s_n in the input model, by definition every s_k along e violates ***AG(¬unsafe)***. Therefore, by construction, every s_k is in *unsafe* and every t_k is in R^p. (1) and (2), and (3) follow immediately.

(<=) Suppose that $e = s_0 t_0 \ldots t_{n-1} s_n$ is an attack in the attack graph G. By construction, all states and transitions of e are also states and transitions in the input model. Since e is an attack, $s_0 \in S_0^p$ and $s_n \in S_s^p$. Therefore, $s_0 \in S_0$ and $s_n \in S$. So e is an execution of the input model, its first state is an initial state of the model, and p is false in its final state. It follows that e violates the property ***AG(¬unsafe)***.

(b) (=>) By construction of the algorithm in Figure 2, all states generated for the attack graph are reachable from an initial state, and all of them violate ***AG(¬unsafe)***. Therefore, for any such state s in the input model, there is a path e_1 from an initial state to s, and there is a path e_2 from s to an *unsafe* state.

The concatenation of e_1 and e_2 is an execution e of the input model that violates ***AG(¬unsafe)***. By Lemma 1a, e is an attack in G. Since e contains s, the proof is complete.

(<=) If there is an attack in G that contains s, then trivially s is in G.

(c) (=>) By lemma 1b, there is an attack $e1 = q_0 t_0 \ldots s_1 \ldots t_{m-1} q_m$ that contains state s_1 and an attack $e2 = r_0 u_0 \ldots s_2 \ldots u_{n-1} r_n$ that contains state s_2. So the following attack includes both states s_1 and s_2 and the transition t: $e = q_0 t_0 \ldots s_1 t s_2 \ldots u_{n-1} r_n$.

(<=) If there is an attack in G that contains t, then trivially t is in G.

–

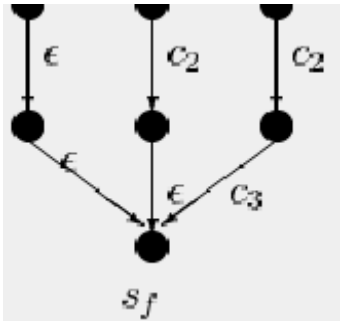


Figure 6. Attack graph corresponding to the set cover problem.

B. NP-Completeness of MCSA

Given an attack graph $G = (S, E, s_0, L)$, consider the problem of finding a minimum critical set of attacks $M(s_0)$. We will call this problem *MCSA* or the minimum critical set of attacks problem. We prove that the decision version of the problem is *NP*-complete.

Lemma 2: Assume that we are given an attack graph $G = (S, E, s_0, L)$ and an integer k. The problem of determining whether there is a critical set $C(s_0)$ such that $|C(s_0)| \leq k$ is *NP*-complete.

Proof: First, we prove that the problem is in *NP*. Guess a set $C \subseteq A$ with size $\leq k$. We need to check that C is a critical set of attacks.

This can be accomplished in polynomial time using the procedure *isCritical (G, C)* described below. Therefore, the problem is in NP.

Next, we prove that the problem is *NP-hard*. The reduction is from the *minimum cover* problem [11, Page 222]. In the minimum cover problem one is given a collection C of subsets of a finite set U and a positive integer $k \leq |C|$. The problem is to determine whether C contains a cover for U of size k or less, i.e., a subset $C' \subseteq C$ with $|C'| \leq k$ such that every element of U belongs to at least one member of C'. We construct an attack graph G_c corresponding to the collection C. The set of attacks A is equal to C. The attack graph G_c has an initial state s_0 and a final state s_f that is unsafe. Let $U = \{u_1,...u_z\}$ and $c_1,...,c_m$ be an enumeration of the collection C. For each collection c_i where $i < m$ we have z new states $s_i1,,...s_{iz}$. There is an edge from s_0 to all the states $s_i1,,...s_{iz}$ corresponding to the collection c_1. There is an edge from $s_{i,j}$ to $s_{i+1,j}$ for all $i < m-1$ and $1 \leq j \leq z$. From each state in the set $\{s_{m-1}1,...,s_{m-1,z}\}$ there as edge to the unsafe state s_f. Label of the edge with tail $s_{i,j}$ is c_i if $u_j \in c_i$, otherwise the label is e. Label of the edge with head $s_{m-1,j}$ is c_m if $u_j \in c_m$ otherwise the label is e. It is easy to prove that there is a critical set of attacks C such that $|C| \leq k$ if and only if there is a cover of size less than or equal to k.

We give a short example to illustrate the reduction. Consider a set $U = \{u_1,u_2,u_3\}$. Suppose that the collection C consists of the following subsets:

$$c_1 = \{u_1, u_2\}$$
$$c_2 = \{u_2, u_3\}$$
$$c_3 = \{u_2\}$$

Notice that there is a cover of size 2, i.e., c_1 and c_2 form a cover. The attack graph corresponding to this problem is shown in Figure 6. The set of attacks is $\{c_1,c_2,c_3\}$. The set of attacks $\{c_1,c_2\}$ is critical because every path from s_0 to the unsafe state uses at least one edge with the label in the set $\{c_1,c_2\}$.

Minimization and Reliability Analyses of Attack Graphs

Somesh Jha1 Oleg Sheyner Jeannette M. Wing

February 2002
CMU-CS-02-109

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Parts of this report will appear in our paper accepted by the IEEE Symposium on Security and Privacy, Oakland, May 2002; and parts are in a paper submitted to the Computer Security Foundations Workshop, Nova Scotia, June 2002

1) Computer Sciences Department, University of Wisconsin, Madison, WI 53706. E-mail: jha@cs.wisc.edu

2) Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213. E-mails: fsheyner,wingg@cs.cmu.edu. This research is sponsored in part by the Defense Advanced Research Projects Agency and the Army Research Office (ARO) under contract no. DAAD19-01-1-0485. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the DOD, ARO, or the U.S. Government.

Abstract

An attack graph is a succinct representation of all paths through a system that end in a state where an intruder has successfully achieved his goal. Today Red Teams determine the vulnerability of networked systems by drawing gigantic attack graphs by hand. Constructing attack graphs by hand is tedious, error-prone, and impractical for large systems. By viewing an attack as a violation of a safety property, we can use model checking to produce attack graphs automatically: a successful path from the intruder's viewpoint is a counterexample produced by the model checker. In this paper we present an algorithm for generating attack graphs using model checking.

Security analysts use attack graphs for detection, defense, and forensics. In this paper we present a minimization technique that allows analysts to decide which minimal set of security measures would guarantee the safety of the system. We provide a formal characterization of this problem: we prove that it is polynomially equivalent to the minimum hitting set problem and we present a greedy algorithm with provable bounds. We also present a reliability technique that allows analysts to perform a simple cost-benefit analysis depending on the likelihoods of attacks. By interpreting attack graphs as Markov Decision Processes we can use a standard MDP value iteration algorithm to compute the probabilities of intruder success for each attack the graph.

We illustrate our work in the context of a small example that includes models of a firewall and an intrusion detection system.

As networks of hosts continue to grow, evaluating their vulnerability to attack becomes increasingly more important to automate. When evaluating the security of a network, it is not enough to consider the presence or absence of isolated vulnerabilities. A large network builds upon multiple platforms and diverse software packages and supports several modes of connectivity. Inevitably, such a network will contain security holes that have escaped notice of even the most diligent system administrator.

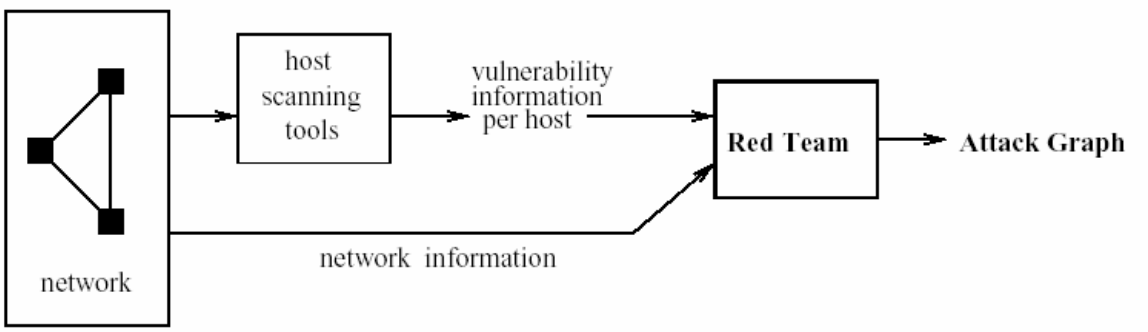


Figure 1: Vulnerability Analysis of a Network

To evaluate the vulnerability of a network of hosts, a security analyst must take into account the effects of interactions of local vulnerabilities and find global vulnerabilities introduced by interconnections. A typical process for vulnerability analysis of a network is shown in Figure 1. First, scanning tools determine vulnerabilities of individual hosts. Using this local vulnerability information along with other information about the network, such as connectivity between hosts, the analyst produces an *attack graph*. Each path in an attack graph is a series of exploits, which we call *atomic attacks*, that leads to an undesirable state (e.g., a state where an intruder has obtained administrative access to a critical host).

1.1 Attack Graphs and Intrusion Detection

Attack graphs can serve as a basis for detection, defense, and forensic analysis. To motivate our study of the generation and analysis of attack graphs, we discuss the potential applications of attack graphs to these areas of security.

Detection

System administrators are increasingly deploying intrusion detections systems (IDSs) to detect and combat attacks on their network. Such systems depend on software sensor modules that first detect suspicious events and activity and then issue alerts. Setting up the sensors involves a trade-off between sensitivity to intrusions and the rate of false alarms in the alert stream. When the sensors are set to report all suspicious events, the sensors frequently issue alerts for benign background events. Frequent false alarms results in administrators turning off the IDS entirely. On the other hand, decreasing sensor sensitivity reduces their ability to detect real attacks.

To address this trade-off, many intrusion detection systems employ heuristic algorithms to correlate alerts from a large pool of heterogeneous sensors. Valdes and Skinner [VS01] describe a probabilistic approach to alert correlation. Successful correlation of multiple alerts increases the chance that the suspicious activity indicated by the alerts is in fact malicious.

Attack graphs can enhance both heuristic and probabilistic correlation approaches. Given a graph describing all likely attacks (i.e., sequences of attacker actions), an IDS can match individual alerts to attack edges in the graph. Matching successive alerts to individual paths in the attack graphs dramatically increases the likelihood that the network is under attack. This on-line vigilance allows the IDS to predict attacker goals, aggregate alarms to reduce the volume of alert information to be analyzed, and reduce the false alarm rates. Knowledge of attacker goals and likely next steps helps guide defensive response.

In this paper we show how to generate attack graphs automatically from models of the network; our models are expressive enough to reflect the administrator's choice of security policy for an IDS and his choice of network configuration. Attack graphs enable an administrator to perform several kinds of analyses to assess their security needs: marking the paths in the attack graph that an IDS will detect; determining where to position new IDS components for best coverage; exploring trade-offs between different security policies and between different software/hardware configurations; and identifying the worst-case scenarios and prioritizing defense strategy accordingly.

Forensics

After a break-in, forensic analysis is used to find probable attacker actions and to assess damage. If legal action is desired, analysts seek evidence that a sequence of sensor alerts comprises a coherent attack plan, and is not merely a series of isolated, benign events. This task becomes even harder when the intruders obfuscate attack steps by slowing down the pace of the attack and varying specific steps. We can construct a convincing argument as to the malicious intent of intruder actions by matching data extracted from IDS logs to a formal reference model based on attack graphs [Ste].

Given that attack graphs can be used to perform a variety of analysis, we can use them to answer the following kinds of questions, of particular interest to system administrators:

Question 1: What successful attacks are undetected by the IDS?

Question 2: If all measures for protecting a network are deployed, does the system become safe?

Question 3: Given a set of measures M, what is the smallest subset of measures M^1 whose deployment makes the system safe?

Answers to these questions, can help a system or network administrator choose the best upgrade strategy. We address these questions in Section 5.

When we are modeling a system operating in an unpredictable environment, certain transitions in the model represent the system's reaction to changes in the environment. We can think of such transitions as being outside of the system's control—they occur when triggered by the environment. When no empirical information is available about the relative likelihood of such environment-driven transitions, we can model them only as nondeterministic "choices" made by the environment. Moreover, for new vulnerabilities data for estimating likelihoods might not be available. However, sometimes empirical data make it possible to assign probabilities to environment-driven transitions. We would like to take advantage of such quantitative information added appropriately to attack graphs. In this context, a system administrator might be interested in answering the following question:

Question 4: The deployment of which security measure(s) will increase the likelihood of thwarting an attacker?

The system administrator can use the answer to question 4 to perform a quantitative evaluation of various security fixes. We address this question in Section 6.2.

1.2 Our Contributions

Constructing attack graphs is a crucial part of performing vulnerability analysis of a network of hosts. Currently, *Red Teams* produce attack graphs by hand, often drawing gigantic diagrams on floor-to-ceiling whiteboards. Doing this by hand is tedious, error-prone, and impractical for attack graphs larger than a hundred nodes.

The main contributions of our work, some of which have appeared in an earlier paper [SHJ_02] are:

- We demonstrate how model checking can be applied to generate attack graphs automatically. We show that the attack graphs produced by our method are *exhaustive*, i.e., covering all possible attacks, and *succinct*, i.e., containing only relevant states and transitions (see Section 3.2).

- Each state transition corresponds to a single atomic attack by the intruder. A state in the model represents the state of the system between atomic attacks. A typical transition from state s_1 to state s_2 corresponds to an atomic attack whose preconditions are satisfied in s_1 and whose effects hold

in state s_2. An *attack* is a sequence of state transitions culminating in the intruder achieving his goal. The entire attack graph is thus a representation of all the possible ways in which the intruder can succeed.

- We prove that finding a *minimum* set of atomic attacks that must be removed to thwart an intruder is NP-complete. Beyond the proof sketched in our earlier paper [SHJ_02], here we further explore the complexity of this problem. Section 5.2.1 proves that the problem is polynomially equivalent to the minimum hitting set problem where the collection of sets is represented as a labeled directed graph. This reduction provided us with additional insight, enabling us to find a greedy algorithm with provable bounds, which can be used to answer questions 1, 2, and 3.

- We present an algorithm to compute the *reliability*-defined as the likelihood of an intruder not succeeding– of a networked system. An advantage of our algorithm is that it allows *incomplete information*, i.e., probabilities of all transitions need not be provided. To our knowledge, previous metrics in the area of security require complete information. We can use this algorithm an answer question 4 precisely.

We present related work in Section 2. Section 3 describes our model and our algorithm to generate attack graphs. We give details of an example networked system in Section 4 and use it throughout the paper for illustrative purposes. In Section 5 we present a minimization analysis to help administrators decide what measures to deploy to thwart attacks. In Section 6 we present a reliability analysis over *probabilistic attack graphs* based on the value iteration algorithm defined for Markov Decision Processes; this analysis can help administrators determine how deployment of one measure can decrease the likelihood of certain attacks. Finally, we present a brief summary and directions for future work in Section 7.

2 Related Work

Phillips and Swiler [PS98] propose a concept of attack graphs similar to the one we describe. However, they model only attacks. Since we have a generic state machine model, we can simultaneously model not just attacks, but also seemingly benign system events (e.g., link failures and user errors) and even system administrator recovery actions. Therefore, our attack graphs are more general than the one proposed by Phillips and Swiler. They also built a tool for generating attack graphs [SPEC00]; it constructs the attack graph by forward exploration starting from the initial state. In our work, we use a symbolic model checker (i.e., NuSMV) that works backward from the goal state to construct the attack graph. A major advantage of the backward algorithm is that vulnerabilities that are not relevant to the safety property (or the goal of the intruder) are never explored; this technique can result in significant savings in space. In fact, Swiler *et al.* [SPEC00] refer to the advantages of the backward search in their paper. Finally, the post-facto analysis suggested by Phillips and Swiler is also different from the ones we present in this paper. We plan to incorporate their analysis into our tool suite.

Dacier [Dac94] proposes the concept of privilege graphs, where each node represents a set of privileges owned by the user and arcs represent vulnerabilities. Privilege graphs are then explored to construct attack state graphs, which represent different ways in which an intruder can reach a certain goal, such as root access on a host. Dacier proposes a metric, called the *mean effort to failure* or METF, based on the attack state graphs. Orlato *et al.* [ODK99] describe an experimental evaluation of this framework. At the surface our notion of attack graphs seems similar to Dacier's. However, as in the case with Phillips and Swiler, Dacier takes an "attack-centric" view of the world; again, our attack graphs are more general. From the experiments conducted by Orlato *et al.* it appears that even for small examples the space required to construct attack state graphs becomes prohibitive. Model checking has made significant advances in representing large state spaces. Therefore, by basing our algorithm on model checking we leverage off those advances and can hope to represent large attack graphs.

Ritchey and Amman [RA01] also used model checking for vulnerability analysis of networks. They used the unmodified model checker SMV [SMV]. Therefore, they could only obtain one counter-example or one attack corresponding to a intruder's goal. In contrast, we modified the model checker

NuSMV to produce complete attack graphs, which represents all possible attacks. We also described analyses that can be performed on these attack graphs. These analyses cannot be meaningfully performed on single attacks.

3 Generating Attack Graphs using Model Checking

First, we formally define *attack graphs*, the data structure used to represent all possible attacks on our networked system. We restrict our attention to attack graphs representing violations of safety properties[1].

Definition 1 Let *AP* be a set of atomic propositions. An *attack graph* or *AG* is a tuple $G = (S, \tau, S_0, S_s, L)$, where S is a set of states, $\tau \subseteq S \times S$ is a transition relation, $S_0 \subseteq S$ is a set of initial states, $S_s \subseteq S$ is a set of success states, and $L: S \rightarrow 2^{AP}$ is a labeling of states with a set of propositions true in that state.

Unless stated otherwise, we assume that the transition relation τ is total. We define an *execution fragment* as a finite sequence of states $s_0 s_1 \ldots s_n$ such that $(s_i, s_{i+1}) \in \tau$ for all $0 \leq i \leq n$. An execution fragment with $s_0 \in S_0$ is an *execution*, and an execution whose final state is in S_s is an *attack*, i.e., the execution corresponds to a sequence of atomic attacks leading to the intruder's goal state. Intuitively, S_s denotes all states where the intruder has achieved his goal, e.g., obtaining root access on a critical host.

Next we turn our attention to algorithms for automatic generation of attack graphs and properties that we can guarantee of them. Starting with a description of a network model *M* and a security property *p*, the task is to construct an attack graph representing all executions of *M* that violate *p*. These are the successful attacks. For the kinds of attack graph analyses suggested in Section 1, it is essential that the graphs produced by the algorithms be *exhaustive* and *succinct*. An attack graph is exhaustive with respect to a model *M* and correctness property *p* if it covers all possible attacks in *M* leading to a violation of *p*, and succinct if it only contains those states and transitions of *M* that lead to a state violating *p*.

3.1 Reachability Analysis

If we restrict ourselves to safety properties, an attack graph may be constructed by performing a simple statespace search. Starting with the initial states of the model *M*, we use a graph traversal procedure (e.g., depth first search) to find all reachable *success* states where the safety property *p* is violated. The attack graph is the union of all paths from initial states to success states.

While this algorithm has the advantage of simplicity, it handles only safety properties and may run into the state explosion problem for non-trivial models. Model [1]checking has dealt with both of these issues with some success, so we will consider algorithms based on that technology.

3.2 Model Checking Algorithm

Model checking is a technique for checking whether a formal model *M* of a system satisfies a given property *p*. In our work, we use the model checker NuSMV [NuS], for which the model *M* is a finite labeled transition system and *p* is a property expressed in *Computation Tree Logic* (*CTL*). For now, we consider only safety properties, which in CTL have the form ***AGf*** (*i.e., p = **AGf***, where *f* is a formula in propositional logic). If the model *M* satisfies the property *p*, NuSMV reports "true." If *M* does not satisfy *p*, NuSMV produces a *counter-example*. A single counter-example shows an execution that leads to a violation of the property. In this section, we explain how to construct attack graphs for safety properties using model checking.

[1] We say more on liveness properties in Section 7.

$S_0 \subseteq S$ – set of initial states
$L : S \rightarrow 2^{AP}$ – labeling of states with propositional formulas
$p = \mathbf{AG}(\neg unsafe)$ (a safety property)

Output:

attack graph $G_p = (S_{unsafe}, R^p, S_0^p, S_f^p, L)$

Algorithm: $GenerateAttackGraph(S, R, S_0, L, p)$

(* Use model checking to find the set of states S_{unsafe} that violate the safety property $\mathbf{AG}(\neg unsafe)$. *)

$S_{unsafe} = modelCheck(S, R, S_0, L, p)$.

(* Restrict the transition relation R to states in the set S_{unsafe} *)

$R^p = R \cap (S_{unsafe} \times S_{unsafe})$.

$S_0^p = S_0 \cap S_{unsafe}$.

$S_f^p = \{s | s \in S_{unsafe} \wedge s \models unsafe\}$.

$return(S_{unsafe}, R^p, S_0^p, S_f^p, L)$.

Figure 2: Algorithm for Generating Attack Graphs

Attack graphs depict ways in which the system can reach an unsafe state (or, equivalently, a successful state for the intruder). We can express the property that an unsafe state cannot be reached as:

AG(¬unsafe)

When this property is false, there are unsafe states that are reachable from the initial state. The precise meaning of *unsafe* depends on the application. For example, in the network security example given in Section 4, the property given below is used to express that the privilege level of the intruder on the host with index 2 should always be less than the root (administrative) privilege.

AG*(network.adversary.privilege[2] < network.priv.root*)

We briefly describe the algorithm (see Figure 2) for constructing attack graphs for the property ***AG(¬unsafe)***. The first step is to determine the set of states S_r that are reachable from the initial state. Next, the algorithm computes the set of reachable states S_{unsafe} that have a path to an unsafe state. The set of states S_{unsafe} is computed using an iterative algorithm derived from a fix-point characterization of the ***AG*** operator [CGP00]. Let R be the transition relation of the model, *i.e., (s,s') Є R* if and only if there is a transition from state s to s'. By restricting the domain and range of R to S_{unsafe} we obtain a transition relation R^p that represents the edges of the attack graph. Therefore, the attack graph is $(S_{unsafe}, R^p, S^p_0, S^p_f, L)$, where S_{unsafe} and R^p represent the set of nodes and edges of the graph respectively; $S^p_0 = S_0 \cap S_{unsafe}$ is the set of initial states; and $S^p_f = \{s \mid s \in S_{unsafe} \wedge s = unsafe\}$ is the set of success states.

In symbolic model checkers, such as NuSMV, the transition relation and sets of states are represented using BDDs [Bry86], a compact representation for boolean functions. There are efficient BDD algorithms for all operations used in the algorithm shown in Figure 2.

3.3 Attack Graph Properties

We can show that an attack graph G generated by the algorithm in Figure 2 is *exhaustive* (Lemma 1(a)) and *succinct* with respect to states and transitions (Lemmas 1(b) and 1(c)).

Lemma 1

(a) (Exhaustive) An execution e of the input model (S, R, S_0, L) violates the property $\rho = \mathbf{AG}(\neg unsafe)$ if and only if e is an attack in the attack graph $G = (Sunsafe, R^p, S_0^p, S_s^p)$.

(b) (Succinct states) A state *s* of the input model *(S, R, S_0, L)* is in the attack graph *G* if and only if there is an attack in *G* that contains *s*.
(c) (Succinct transitions) A transition *t* = (s1, s2) of the input model *(S, R, S_0, L)* is in the attack graph *G* if and only if there is an attack in *G* that includes *t*.

Proof:

(a) exhaustive. (=>) Let $e = s_0 t_0 \ldots t_{n-1} s_n$ be a (finite) execution of the input model such that s_n is an *unsafe* state. To prove that *e* is an attack in *G*, it is sufficient to show (1) $s_0 \in S_0^p$, (2) $s_n \in S_s^p$, and (3) for all $0 \leq k \leq n$, $s_k \in S$ and $t_k \in R^p$.

Since *unsafe* holds at s_n and for all *k* there is a path from s_k to s_n in the input model, by definition every s_k along *e* violates ***AG(¬unsafe)***. Therefore, by construction, every s_k is in S_{unsafe} and every t_k is in R^p. (1) and (2), and (3) follow immediately.

(<=) Suppose that $e = s_0 t_0 \ldots t_{n-1} s_n$ is an attack in the attack graph *G*. By construction, all states and transitions of *e* are also states and transitions in the input model. Since *e* is an attack, $s_0 \in S_0^p$ and $s_n \in S_s^p$. Therefore, $s_0 \in S_0$ and $s_n \in S$. So *e* is an execution of the input model, its first state is an initial state of the model, and *p* is false in its final state. It follows that *e* violates the property ***AG(¬unsafe)***.

(b) succinct state (=>) By construction of the algorithm in Figure 2, all states generated for the attack graph are reachable from an initial state, and all of them violate ***AG(¬unsafe)***. Therefore, for any such state *s* in the input model, there is a path e_1 from an initial state to *s*, and there is a path e_2 from *s* to an *unsafe* state.

The concatenation of e_1 and e_2 is an execution *e* of the input model that violates ***AG(¬unsafe)***. By Lemma 1a, *e* is an attack in *G*. Since *e* contains *s*, the proof is complete.

(<=) If there is an attack in *G* that contains *s*, then trivially *s* is in *G*.

(C) Succinct-transition. (=>) By lemma 1b, there is an attack $e_1 = q_0 t_0 \ldots s_1 \ldots t_{m-1} q_m$ that contains state s_1 and an attack $e2 = r_0 u_0 \ldots s_2 \ldots u_{n-1} r_n$ that contains state s_2. So the following attack includes both states s_1 and s_2 and the transition *t:* $e = q_0 t_0 \ldots s_1 t s_2 \ldots u_{n-1} r_n$.

(<=) If there is an attack in *G* that contains *t*, then trivially *t* is in *G*.

4 A Simple Intrusion Detection Example

Consider the example network shown in Figure 3. There are two target hosts, ip_1 and ip_2, and a firewall separating them from the rest of the Internet. As shown, each host is running two of three possible services (ftp, sshd, a database). An intrusion detection system (IDS) monitors the network traffic between the target hosts and the outside world. There are four possible atomic attacks, identified numerically as follows: (0) sshd buffer overflow, (1) ftp .rhosts, (2) remote login, and (3) local buffer overflow. If an atomic attack is *detectable*, the intrusion detection system will trigger an alarm; if an attack is *stealthy*, the IDS misses it. The ftp .rhosts attack needs to find the target host with two vulnerabilities: a writable home directory and an executable command shell assigned to the ftp user name. The local buffer overflow exploits a vulnerable version of the xterm executable.

In this section, we construct a finite state model of the example network so that each state transition corresponds to a single atomic attack by the intruder. A state in the model represents the state of the system between atomic attacks. A typical transition from state s_1 to state s_2 corresponds to an atomic attack whose preconditions are satisfied in s_1 and whose effects hold in state s_2.

The intruder launches his attack starting from a single computer, ip_a, which lies outside the firewall. His eventual goal is to disrupt the functioning of the database. For which, the intruder needs root access on the database host ip_2.

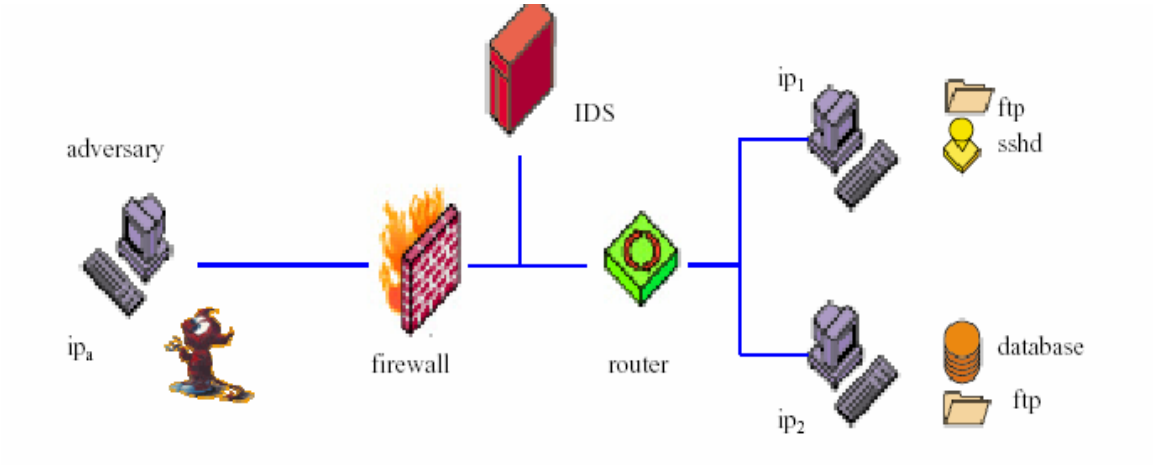


Figure 3: Example Network

4.1 States of the Finite State Machine Model

The Network

We model the network as a set of facts, each represented as a relational predicate. The state of the network specifies services, host vulnerabilities, connectivity, and a remote login trust relationship between hosts. There are six boolean variables for each host, specifying whether any of the three modeled services are running and whether any vulnerabilities are present on that host.

variable	meaning
ssh_h	ssh service is running on host h
ftp_h	ftp service is running on host h
$data_h$	database is running on host h
$wdir_h$	ftp home directory is writable on host h
$fshell_h$	ftp user has executable shell on host h
$xterm_h$	xterm executable is vulnerable to overflow on host h

Connectivity is expressed as a ternary relation $R \subseteq Host \times Host \times Port$, where $R(h_1, h_2, p)$ means that host h_2 is reachable from host h_1 on port p. The constants sp and fp will refer to the specific ports for the ssh and ftp services, respectively. Slightly abusing notation (by overloading R), we write $R(h_1, h_2)$ when there is a network route from h_1 to h_2. We model trust as a binary relation $RshTrust \subseteq Host \times Host$, where $RshTrust(h_1, h_2)$ indicates that a user may log in from host h_2 to host h_1 without authentication (i.e., host h_1 "trusts" host h_2).

The Intruder

The function $plvl_A: Hosts \rightarrow \{none, user, root\}$ gives the level of privilege that intruder A has on each host. There is a total order on the privilege levels: $none < user < root$.

Several state variables specify which attack the intruder will attempt next:

variable	meaning
attack	attack type
source	source host
target	target host
strain	stealthy/detectable attack

If detectable, it will trigger an alarm when executed on a host or network segment monitored by the IDS; if an attack is *stealthy*, the IDS does not detect it.

We specify the IDS with a function *ids: Host x Host x Attack → {d, s, b}*, where $ids(h_1, h_2, a) = d$ if attack a is *d*etectable when executed with source host h_1 and target host h_2; $ids(h_1, h2, a) = s$ if attack *a* is *s*tealthy when executed with source host *h1* and target host h_2; and $ids(h_1, h2, a) = b$ if attack a has *b*oth detectable and stealthy strains, and success in detecting the attack depends on which strain is used. When *h1* and h_2 refer to the same host, $ids(h_1, h2, a)$ specifies the intrusion detection system component (if any) located on that host. When h_1 and h_2 refer to different hosts, $ids\ ids(h_1, h2, a)$ specifies the intrusion detection system component (if any) monitoring the network path between h_1 and h_2. In addition, a global boolean variable specifies whether the IDS alarm has been triggered by any previously executed atomic attack.

4.2 Initial States

Initially, there is no trust between any of the hosts; the trust relation *Tr* is empty. The connectivity relation *R* is shown in the following table. An entry in the table corresponds to a pair of hosts (h_1, h_2). Each entry is a triple of boolean values. The first value is 'y' if h_1 and h_2 are connected by a physical link, the second value is 'y' if h_1 can connect to h_2 on the ftp port, and the third value is 'y' if h_1 can connect to h_2 on the sshd port.

R	ip_a	ip_1	ip_2
ip_a	y,n,n	y,y,y	y,y,n
ip_1	y,n,n	y,y,y	y,y,n
ip_2	y,n,n	y,y,y	y,y,n

We use the connectivity relation to reflect the firewall rule sets as well as the existence of physical links. For the table above, the firewall is open and does not place any restrictions on the flow of network traffic.

Initially, the intruder has *root* privileges on his own machine ip_a and no privileges on the other hosts.

The paths between (ip_a, ip_1) and between (ip_a, ip_2) are monitored by the single network-based IDS. The path between (ip_1, ip_2) is not monitored. There are no other host-based intrusion detection components. The IDS detects the remote login attack and the detectable strains of the sshd buffer overflow attack.

4.3 Transitions

Our model has nondeterministic state transitions. If the current state of the network satisfies the **preconditions** of more than one atomic attack rule, the intruder nondeterministically "chooses" one of those attacks. The state then changes according to the **effects** clause of the chosen attack rule. The intruder repeats this process until his goal is achieved.

We model four atomic attacks. Throughout the description, *S* is used to designate the source host and *T* the target host. Recall that *R(S, T, p)* denotes that host *T* is reachable from host *S* on port *p*.

Sshd Buffer Overflow

This remote-to-root attack immediately gives a remote user a root shell on the target machine.

intruder preconditions	
$plvl_A(S) \geq$ user	*User-level privileges on host S*
$plvl_A(T) <$ root	*No root-level privileges on host T*
network preconditions	
ssh_T	*Host T is running sshd*
$R(S, T, sp)$	*Host T is reachable from S on port* sp
intruder effects	
$plvl_A(T) =$ root	*Root-level privileges on host T*
network effects	
$\neg\text{ssh}_T$	*Host T is not running sshd*
end	

Ftp .rhosts

Using an tp vulnerability, the intruder creates an .rhosts file in the ftp home directory, creating a remote login trust relationship between his machine and the target machine.

attack ftp-rhosts **is**	
intruder preconditions	
$plvl_A(S) \geq$ user	*User-level privileges on host S*
network preconditions	
ftp_T	*Host T is running ftp*
$R(S, T, fp)$	*Host T is reachable from S on port* fp
wdir_T	*Ftp directory writable on host T*
fshell_T	*Ftp user has been assigned a valid shell on host T*
$\exists X.\neg RshTrust(X, T)$	*No rsh trust for some host X and T*
intruder effects	
none	
network effects	
$\forall X.RshTrust(X, T)$	*Rsh trust between all hosts and T*
end	

Remote Login

Using an existing remote login trust relationship between two machines, the intruder logs in from one machine to another, getting a user shell without supplying a password. This operation is usually a legitimate action performed by regular users, but from the intruder's viewpoint, it is an atomic attack.

attack rsh-login **is**
  intruder preconditions
    $plvl_A(S) = \text{user}$ — *User-level privileges on host S*
    $plvl_A(T) = \text{none}$ — *No privileges on host T*
  network preconditions
    $RshTrust(S, T)$ — *Rsh trust between S and T*
    $R(S, T)$ — *Host T is reachable from S*
  intruder effects

Local Buffer Overflow

If the intruder has acquired a user shell on the target machine, the next step is to exploit a buffer overflow vulnerability on a setuid root file to gain root access.

attack local-setuid-buffer-overflow **is**
  intruder preconditions
    $plvl_A(T) = \text{user}$ — *User-level privileges on host T*
  network preconditions
    xterm_T — *There is a vulnerable xterm executable*
  intruder effects
    $plvl_A(T) = \text{root}$ — *Root-level privileges on host T*
  network effects
    none
end

It is easy to see that each atomic attack strictly increases either the intruder's privilege level on the target host or remote login trust between hosts. This means that *the attack graph has no cycles*.

From our finite model we can now automatically construct attack graphs that demonstrate how the intruder can violate various security properties. Suppose we want to generate all attacks that demonstrate how the intruder can gain root privilege on host ip_2 and remain undetected by the IDS. The following CTL formula expresses the safety property that *the intruder on host ip_2 always has privilege level below root* or *is detected*:

AG(*network.adversary.privilege[2] < network.priv.root* | *network.detected*)

Figure 4 shows the attack graph produced by our tool for this property. Each node is labeled by an attack id number, which corresponds to the atomic attack *to be attempted next*; a flag S/D indicates whether the attack is stealthy or detectable by the intrusion detection system; and the numbers of the source and target hosts (ip_a corresponds to host number 0).

Any path in the graph from the root node to a leaf node shows a sequence of atomic attacks that the intruder can employ to achieve his goal while remaining undetected. For instance, the path highlighted by dashed-boxed nodes consists of the following sequence of four atomic attacks: overflow sshd buffer on host 1, overwrite .rhosts file on host 2 to establish rsh trust between hosts 1 and 2, log in using rsh from host 1 to host 2, and finally, overflow a local buffer on host 2 to obtain root privileges.

We have also expanded the example described above by adding two additional hosts, four additional atomic attacks, several new vulnerabilities, and flexible firewall configurations. For this larger example the attack graph has 5948 nodes and 68364 edges.

5 Minimization Analysis

Once we have an attack graph generated for a specific network with respect to a given safety property, we can utilize it for further analysis. A system administrator has available to him a set of *measures*, such as deploying additional intrusion detection tools, adding firewalls, upgrading software, deleting user accounts,

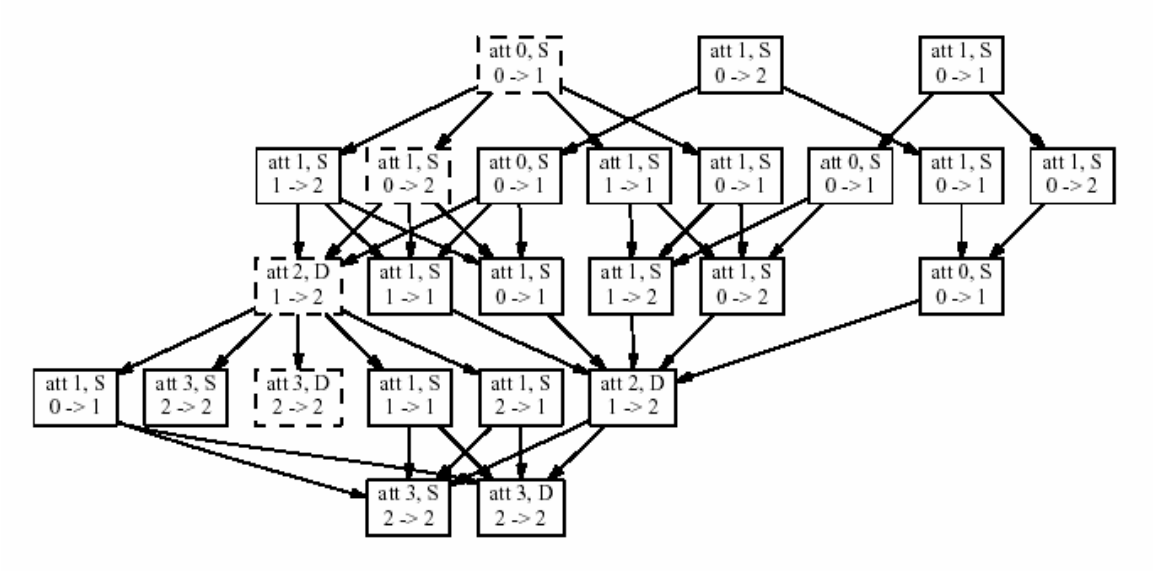


Figure 4: Attack Graph

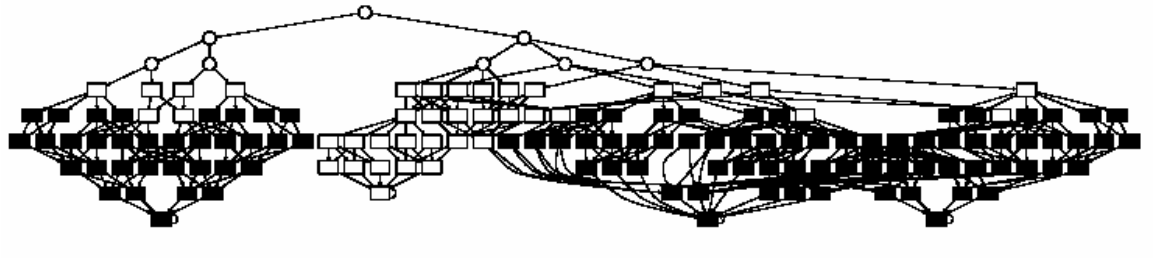


Figure 5: Attack Graph Analysis

Minimization analysis helps analysts make decisions about what measures to deploy depending on what set of atomic attacks they thwart. It helps us answer questions such as 1, 2, and 3 posed in Section 1.1. Let us look at each question in turn since they suggest different solution approaches.

5.1 Minimal Subsets of Atomic Attacks to Thwart

Suppose we want to find a minimal set, A, of atomic attacks that must be prevented to guarantee the adversary cannot achieve his goal. A system analyst can use this information in deciding to choose one measure m_1, which eliminates this minimal set of attacks over another measure, m_2, perhaps cheaper than m_1, but ineffective with respect to A.

A naive solution is as follows:

1. Make only a subset of the atomic attacks available to the intruder.

2. Run the model checking algorithm to determine if the adversary can succeed.

3. Do Steps 1 and 2 for all possible non-empty subsets of atomic attacks.

Clearly this solution is exponential in the number of atomic attacks. For our example, however, the number is small, and we can easily determine this minimal set. As a by-product of determining this set, we can easily answer the first question posed in Section 1.

Question 1: What successful attacks are undetected by the IDS?

Answer: To answer this question, we modify the model slightly. For simplicity, we nondeterministically decide which subset to consider initially, before any attack begins; once the choice is made, the subset of available atomic attacks remains constant during any given attack. We ran the model checker on the modified model with the invariant property that says the intruder never obtains root privilege on host ip_2:

$$\mathbf{AG}(network.adversary.privilege[2] < network.priv.root)$$

The post-processor marked the states where the intruder has been detected by the IDS. The result is shown in Figure 5. The white rectangles indicate states where the attacker had not yet been detected by the intrusion detection system. The black rectangles are states where the intrusion detection system has sounded an alarm. Thus, white leaf nodes are desirable for the attacker because his objective is achieved without detection. Black leaf nodes are less desirable. The attacker achieves his objective, but the alarm goes off.

The resolution of which atomic attacks are available to the intruder happens in the circular nodes near the root of the graph. The first transition out of the root (initial) state picks the subset of attacks that the intruder will use. Each child of the root node is itself the root of a disjoint subgraph where the subset of atomic attacks chosen for that child is used. Note that the number of such subgraphs descending from the root node corresponds to the number of subsets of atomic attacks with which the intruder can be successful. The model checker determines that for any other possible subset, there is no possible successful sequence of atomic attacks.

The root of the graph in Figure 5 has two subgraphs, corresponding to the two subsets of atomic attacks that will allow the intruder to succeed. In the left subgraph the sshd buffer overflow attack is not available to the intruder; it can be readily seen that the intruder can still succeed, but cannot do so while remaining undetected by the IDS. In the right subgraph, all attacks are available. Thus, the entire attack graph implies that all atomic attacks other than the sshd attack are indispensable: the intruder cannot succeed without them. That is, for no other subset of atomic attacks can the intruder succeed in achieving his goal. The analyst can use this information to guide decisions on which network defenses can be profitably upgraded.

The white cluster in the middle of the figure is isomorphic to the attack graph presented in Figure 4; it shows attacks in which the intruder can achieve his objective without detection (i.e., all paths by which the intruder reaches a white leaf in the graph).

$$\mathbf{AG}(\neg unsafe)$$

Let A be the set of atomic attacks, and $G = (S, E, s_0, s_s, L)$ be the attack graph, where S is the set of states, $E \subseteq S \times S$ is the set of edges, $s_0 \in S$ is the initial state, $s_s \in S$ is the success state for the intruder, and $L: E \rightarrow A \cup \{\varepsilon\}$ is a labeling function where $L(e) = a$ if an edge $e = (s \rightarrow s')$ corresponds to an atomic attack a, otherwise $L(e) = e$. Edges labeled with e represent system transitions that do not correspond to an atomic attack. Moreover, as demonstrated below additional e edges can be also introduced by our construction. Without loss of generality we can assume that there is a single initial and success state. For

example, consider an attack graph with multiple initial states $s_0^1 \ldots s_0^j$ and success states $s_s^1 \ldots s_u^s$. We can add a new initial state s_0 and a new success state s_s with *e*-labeled edges (s_0, s_0^m) $(1 \leq m \leq j)$ and $(s_s, s_s^t)(1 \leq t \leq u)$.

Suppose we are also given a finite set of measures $M = (m_1, \ldots, m_k)$ and a function $covers: M \rightarrow 2^A$. An atomic attack $a \in covers(m_i)$ if adopting measure m_i removes the atomic attack a.

We are now ready to address the question of what measures a system administrator should deploy to ensure the system is safe. Again, there is a naive solution, that is, to try all possible subsets of measures $M' \subseteq M$ and determine which of those make the system safe. We discuss this approach in the context of question 2:

Question 2: If all measures for protecting a network are deployed, does the system become safe?

Answer: A network administrator wants to find out whether adopting measures from a set $M' \subseteq M$ will make the network safe. This question can be answered in linear time using the attack graph G. First, we define $covers(M')$ as $U_{m \in M'} covers(m)$. Next, we remove all edges e from G such that $L(e) \in covers(M')$. The network is safe iff the success state s_s is not reachable from the initial state s_0. This simple reachability question can be answered in time that is linear in the size of the graph.

As the set of measures grows (and as the set of atomic attacks grows), we really would like to have the system administrator choose the smallest subset of measures that would guarantee the networked system is safe. We address this decision in the context of question 3:

Question 3: Given a set of measures M, what is the smallest subset of measures M' whose deployment makes the system safe?

Answer: A network administrator wishes to find a subset $M' \subseteq M$ of smallest size, such that adopting the measures in the set M' will make the network safe. Unfortunately, this problem is *NP*-complete, but we develop good approximation algorithms. We proceed in two steps:

Step 1: Finding a small set of atomic attacks.

In this step, we find a set of atomic attacks whose removal makes the network safe. As described in the previous section, checking every possible subset of attacks is exponential in the number of attacks. In an earlier conference paper [SHJ_02], we show that finding the *minimum* set of atomic attacks which must be removed to thwart an intruder is in fact *NP*-complete. We repeat part of the proof below (see Lemma 2). We also demonstrated how a *minimal* set can be found in polynomial-time.[2] In this paper, we further explore the complexity of this problem. Section 5.2.1 proves that the problem of finding a minimum set of attacks is polynomially equivalent to the minimum hitting set problem, where the collection of sets is represented as labeled directed graph. This reduction provided us with additional insight. This additional insight enabled us to find a greedy algorithm with provable bounds. Recall that $M = (m_1, \ldots, m_k)$ is the set of measures and $covers: M \rightarrow 2^A$ is a function, where $covers(m_i)$ represents the set of atomic attacks that are removed by adopting the measure m_i. With each attack a in the set A', we associate a set of measures $M(a)$ which is $\{m_i \mid a \in covers(m_i)\}$. The set of attacks A' defines a collection $C_{A'}$ of subsets of M. We wish to find the smallest subset $M' \subseteq M$ such that for all $a \in A'$ there exists an $m_i \in M'$ such that $a \in covers(m_i)$, or equivalently $M' \cap M(a) \neq 0$. This is known as the minimum hitting set problem, which is *NP*-complete, but good approximation algorithms exist to solve this problem (see Section 5.2.2)

5.2.1 The Minimum Critical Attack Sets and the Minimum Hitting Set Problem

This section addresses the first step in the answer to question 3. Assume that we are given an attack graph $G = (S, E, s_0, s_s, L)$, where S is the set of states, $E \subseteq S \times S$ is the set of edges, $s_0 \in S$ is the initial state, $s_s \in S$ is the success state for the intruder, and $L : E \rightarrow A \cup \{e\}$ is a labeling function.

Given a state $s \in S$, a set of attacks C is *critical* with respect to s if and only if the intruder cannot reach his goal from s when the attacks in C are removed from his arsenal. Equivalently, C is critical with respect to s if and only if every path from s to the success state s_s has at least one edge labeled with an attack $a \in C$.

A critical set corresponding to a state s is *minimum* (denoted $M(s)$) if there is no critical set $M'(s)$ such that $|M'(s)| < |M(s)|$. In general, there can be multiple minimum sets corresponding to a state s. Of course, all minimum critical sets must be of the same size.

A critical set of an attack graph $G = (S, E, s_0, s_s, L)$ is defined as a critical set corresponding to the initial state s_0. Therefore, the *Minimum Critical Set of Attacks (MCSA) problem* is the problem of finding a minimum critical set of attacks $M(s_0)$. The decision version of the problem is defined as follows: given an attack graph $G = (S, E, s_0, s_s, L)$ and a positive integer K, is there a critical set of attacks $A \subseteq A$ such that $|A| \leq K$?

Lemma 2 Assume that we are given an attack graph $G = (S, E, s_0, s_s, L)$ and an integer k. The MCSA problem of determining whether there is a critical set $C(s_0)$ such that $|C(s_0)| \leq k$ is *NP*-complete.

Proof: *First, we prove that the problem is in* NP. *Guess a set $C \subseteq A$ with size $\leq k$. We need to check that C is a critical set of attacks. This can be accomplished in polynomial time using the reachability algorithm described before (see answer to question 2). Therefore, the problem is in* NP.

Next, we prove that the problem is NP-*hard. The reduction is from the* minimum hitting set *problem, details as given in the remainder of this section.*

Assume that we are given an attack graph $G = (S, E, s_0, s_s, L)$. A *path* π is a sequence of states $q_1, \ldots, q_n$, such that $q_i \in S$ and $(q_i, q_{i+1}) \in E$. A *complete path* starts from the initial state s_0 and ends in the success state s_s. The label of a path $\pi = q_1, \ldots q_n$ (abusing notation, we will denote it also as $L(\pi)$) is a subset of a set of attacks A.

$$U_{i=1}^{n-1} L(q_i, q_{i+1})\} \setminus \{e\}.$$

$L(\pi)$ represents the set of atomic attacks used on the path π. A set of attacks $A \subseteq A$ is called *realizable* in the attack graph G iff there exists a complete path π in G such that $L(\pi) = A$. In other words, an intruder can use the set of attacks A to start from the initial state and reach the success state. The set of all realizable sets in an attack graph G is denoted by $Rel(G)$. The following lemma is easy to prove and follows straight from the definitions.

Lemma 3 Assume that we are give an attack graph $G = (S, E, s_0, s_s, L)$. A set of attacks A is critical iff

$$\forall A' \in Rel(G). A' \cap A \neq 0.$$

In other words, all realizable sets have a non-empty intersection with a critical set A.

Question: Is there a subset $S \subseteq S$ with $(S) \leq K$ such that S contains at least one element from each subset in C?

Lemma 3 proves that the problem of finding whether the attack graph G has a critical set of size $\leq K$ *is the* hitting set problem with $C = Rel(G)$, $S = A$, and K.

Next suppose we have an instance (C, S, K) of the hitting set problem. We will construct an attack graph $G' = (S', E', s_0', s_s', L')$, where $L': E' \rightarrow S \cup \{e\}$, i.e., the set of attacks used in the attack graph G' is S. Moreover, the set of realizable sets $Rel(G')$ of the graph G' is the collection C. A critical set of size $\leq K$ of the attack graph G' is a hitting set for the collection C. Next, we describe the construction of G'. Let $C = \{C_1, \ldots, C_m\}$ be the collection of sets and $S = \{s_1, \ldots, s_n\}$ be the set. We make m copies $S^1, \ldots, S^m$ of the set S. The set of elements in S^i will be denoted by $\{s^i, \ldots, s^i_n\}$. The set of states S' in the attack graph G' is

$$\{s_0', s_s'\} \cup S^1 \cup \ldots \cup S^m.$$

The initial state is s_0' and the final state is s_s'. The set of edges E' and the labeling function L' are defined as follows:

- There is an edge from s'_0 to every state in the set $\{s^1_1, s^2_1, \ldots, s^m_1\}$, and label of the edge (s'_0, s^1_i) is s_i if $s_1 \in C_i$, otherwise it is e.
- For all $1 \leq i \leq m$ and $1 \leq j \leq n-1$, there is an edge (s^i_j, s^i_{j+1}), and the label of edge (s^i_j, s^i_{j+1}) is s_{j+1} *if* $s_{j+1} \in Ci$, otherwise it is e.
- There is an edge from every state in the set $\{s^1_n, s^2_n, \ldots, s^m_n\}$ to the state s'_s, and labels of all these edges is e.

The sizes of the sets S' and E' in the attack graph G' are $mn + 2$ and $2m + mn$ respectively. It is easy to see that $Rel(G')$ is equal to C, and $S' \subseteq S$ is a critical set of the attack graph G' iff S' is a hitting set for the collection C. Since the size of G' is polynomial in the size of the instance of the hitting set problem and the hitting set problem is *NP*-complete, the MCSA problem is *NP*-hard. Lemma 2 proves that MCSA is in *NP*. Therefore, MCSA is *NP*-complete. The next example illustrates our construction.

Note: The discussion above also proves that the problem of finding a minimum set of measures whose adoption will make the network safe is also *N*P-complete. One can simply take the set of measures M to be the set of attacks A.

Example 1 We give a short example to illustrate the reduction. Consider a set $S = \{s_1, s_2, s_3\}$. Suppose that the collection C consists of the following subsets:

$$C_1 = \{s_1, s_2\}$$
$$C_2 = \{s_2, s_3\}$$
$$C_3 = \{s_2\}$$

The attack graph G' corresponding to this problem is shown in Figure 6. The set of attacks is $\{s_1, s_2, s_3\}$. The set of realizable sets $Rel(G')$ is exactly the collection C. The set of attacks $\{s_1, s_2\}$ is critical because every path from s'_0 to the success state s'_s uses at least one edge with the label in the set $\{s_1, s_2\}$. Moreover, $\{s_1, s_2\}$ is a hitting set for the collection $C = \{C_1, C_2, C_3\}$.

The above discussion proves that the problem of finding critical sets in attack graph is *polynomially equivalent* to finding hitting sets for a collection, with one caveat–the collection of sets C is represented as an attack graph. *An attack graph can be an exponentially succinct representation of a collection of sets*. Figure 7 shows an attack graph of linear size whose set of realizable sets is the power set of $\{s_1, \ldots, s_n\}$. Therefore, the minimum critical set problem is polynomially equivalent to the hitting set problem where the collection of sets C is represented as a labeled directed graph.

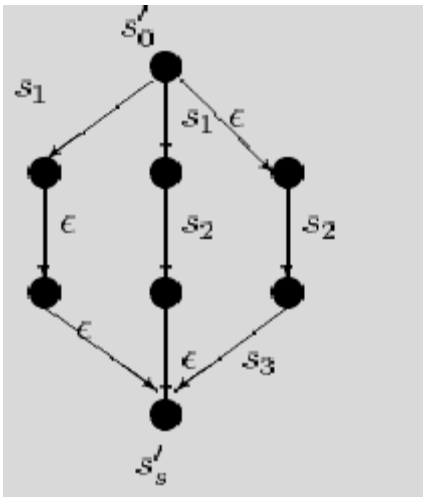


Figure 6: Attack graph corresponding to the collection C.

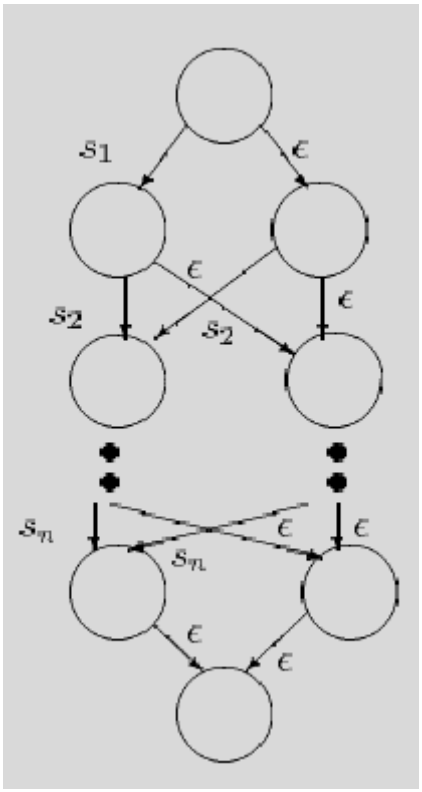


Figure 7: Attack graph representing an exponential number of realizable sets.

Let *(C, S, K)* be an instance of the hitting set problem. Let *S'* and *C'* be initially the empty set. The greedy algorithm executes the following step until *C'= C*.

- Pick an element *s* out of the set *S \ S'* that covers the maximum number of sets in the collection *C \ C'*. An element s is said to cover a set $S_1 \subseteq S$ iff $s \in S_1$.
- Let *s* be the element picked in the previous step and *C(s)* be the collection of sets in *C* covered by *s*. Update *S'* and *C'* as follows:

$$S' \leftarrow S' \cup \{s\}$$
$$C' \leftarrow C' \cup C(s)$$

- Let H_d be the d-th harmonic number $\sum^{d}_{i=1} 1/i$. Let $C(s)$ be the number of sets in the collection C that are covered by the element s.

Lemma 4 *GREEDY-HITTING-SET* is a polynomial-time $p(n)$-approximation algorithm, where $p(n) = H(max_{s \in S}\{|C(s)|\})$.

The proof of the lemma follows from the equivalence between the minimum hitting set and the minimum cover problem [ADP80] and the proof of the approximation factor $p(n)$ for the greedy algorithm for the minimum cover problem [CLR85]. Using the equivalence between the problems of finding a minimum critical set and a minimum hitting set, we can construct a greedy procedure (called *GREEDY-CRITICAL-SET*) for finding a critical set for the attack graph. Assume that we are given an attack graph $G = (S, E, s_0, s_s, L)$, where S is the set of states, $E \subseteq S \times S$ is the set of edges, $s_0 \in S$ is the initial state, $s_s \in S$ is the success state for the intruder, and $L : E \rightarrow A \cup \{e\}$ is a labeling function. Moreover, assume that we can compute in polynomial time the function $\mu_G : A \rightarrow N$, where $\mu_G(a)$ is the number of realizable sets in the attack graph G that contain the attack a. Formally, $\mu_G(a)$ is equal to

$$|\{A' \mid a \in A' \text{ and } A' \in Rel(G)\}|$$

Initially, let A' be the empty set and $G' = G$. The greedy algorithm *GREEDY-CRITICAL-SET* executes the following step until G' is empty.

- Pick and element a from the set $A \setminus A'$ that maximizes $\mu_{G'}(a)$.
- Let a be the element picked in the previous step. Update A' and G' as follows:

  $A' \leftarrow A \cup \{a\}$
  Remove all edges labeled with a from G'

Lemma 5 *GREEDY-CRITICAL-SET* is a polynomial-time $p(n)$-approximation algorithm, where $p(n) = H(max_{a \in A}\{\mu_G(a)\})$.

Next, we explore conditions when the function μ_G can be computed in polynomial time. Assume that the attack graph G is a DAG. An argument for this was given in Section 4.3. Moreover, assume that each atomic attack is *used only once* on a path from the initial state s_0 to the success state s_s. This is not an unreasonable assumption because the attack graph edges are labeled with instantiations of attack templates shown in Section 4.3, e.g., a local-setuid-buffer-overflow attacks on two different hosts are distinct in the attack graph. Such attack graphs are called *use-once* DAGs. The following lemma is easy to prove.

Lemma 6 For an attack graph that is a use-once DAG, the function μ_G can be computed in time that is linear in size of the attack graph.

Suppose a system administrator would like to know which measures would increase or decrease the likelihood of thwarting an attack? If we have probabilities available to us, we can annotate attack graphs to help system administrators answer such questions.

In our work, we do not require that all transitions be given probabilities; in general, our annotated attack graphs can have a mix of probabilistic and nondeterministic state transitions. We pursue the implications of this general kind of attack graph in this section.

In general, we also do not require probabilities to be numeric; they can be symbolic, e.g., "high," "medium," or "low," and even partially ordered. In an earlier paper [JW01], we discuss an analysis that uses symbolic probabilities; in this paper, however, we restrict ourselves to numeric values.

6.1 Probabilistic Attack Graphs

Suppose that the graph has a state s with only two outgoing transitions. In a regular attack graph, the choice of which transition to take when the system is in state s is nondeterministic. However, we may have some empirical data that enables us to estimate that whenever the system is in state s, on average it will take one of the transitions four times out of ten and the other transition six remaining times. We can place probabilities 0.4 and 0.6 on the corresponding edges in the attack graph. Intuitively, the probability of the transition $s \rightarrow s'$ represents the likelihood that the atomic attack corresponding to the transition will succeed. We call a state with known probabilities for outgoing transitions *probabilistic*. When we have assigned all known probabilities in this way, we are left with an attack graph that has some probabilistic and some nondeterministic states in it. We call such mixed attack graphs *probabilistic attack graphs*. We use probabilistic attack graphs to evaluate the reliability of a network. Note that probabilities of all the transitions might not be available because of lack of data, e.g., a new type of atomic attack.

Since the attack graph includes only those states and transitions that can lead to success states, it excludes some transitions that exist in the complete model M. These excluded transitions can have non-zero probability, so that the sum of probabilities of transitions from a probabilistic state will be less than 1. To address this problem, we must model the rest of M in some way. We add a "catch-all" *escape* state s_e to the attack graph. A probabilistic state s in the attack graph will have a transition to s_e if and only if in M there is a transition from s to some state *not* in the attack graph. The probability of going from s to s_e will be 1 minus the sum of the probabilities of going to other states. There are no transitions out of s_e except a self-loop (which preserves the totality of the transition relation τ).

In an attack graph containing the escape state s_e attacks are allowed to terminate in s_e. We will call them *escape attacks*, or attacks that were pre-empted by the intruder before he reached his goal.

6.1.1 Definition of PAGs

Definition 3 A *probabilistic attack graph* or PAG is a tuple $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$, where S_n is a set of nondeterministic *states*, S_q is a set of probabilistic states, $s_e \in S_n$ is a nondeterministic escape state $(s_e \notin S_s)$, $S = S_n \cup S_q$ is the set of all states, $\tau \subseteq S \times S$ is a transition relation, $\pi \rightarrow S_q \rightarrow S \rightarrow R$ are transition probabilities, $S_0 \subseteq S$ is a set of initial states, $S_s \subseteq S$ is a set of success states, and $L : S \rightarrow 2^{AP}$ is a labeling of states with a set of propositions true in that state.

A probabilistic attack graph distinguishes between nondeterministic states (set S_n) and probabilistic states (set S_q). Moreover, the sets of nondeterministic and probabilistic states are disjoint $(S_n \cap S_q = 0)$. The function π specifies probabilities of transitions from probabilistic states, so that for all transitions $s_1 \rightarrow s_2 \in \tau$ such that $s_1 \in S_q$, we have $P(s_1 \rightarrow s_2) = \pi(s1)(s2) > 0$. Thus, $\pi(s)$ can be viewed as a probability distribution on next states. Intuitively, when the system is in a nondeterministic state s_n, we have no information about the relative probabilities of the possible next transitions. When the system is in a probabilistic state s_q, it will choose the next state according to probability distribution $\pi(s_q)$.

Let $G = (S, \tau, S_0, S_s, L)$ be the attack graph and P a function that assigns probabilities to transitions. The probabilities can be loosely interpreted as the probability of the atomic attack corresponding to the transition succeeding. We are interested in finding the reliability of the attack graph, i.e., the probability that the intruder will not succeed. We can view G as a Markov chain with S as its state space and $P(s_1 \rightarrow s_2)$ as its transition probability. Let $U : S \rightarrow R^+$ be the steady state probability of the Markov chain (see [Dur95] for definitions and technical conditions). In this case, the reliability of the attack graph G is given by the following expression:

$$1 - \sum_{s \in S_s} U(s)$$

In other words, the reliability is the probability that in the "long run" the Markov chain will not be in a state in the set S_s.

In general, however, we do not have probabilities assigned to all transitions; thus in Section 6.2 we show how to perform similar reliability analysis on probabilistic attack graphs in the presence of

nondeterministic states. The justification of our approach relies on converting a probabilistic attack graph (PAG) into an alternating probabilistic attack graph (APAG) and then interpreting the result as a Markov Decision Process; we give this construction and interpretation in Section 6.3; we give the proof of correctness of the MDP value iteration algorithm applied to PAGs in Section 6.4. Sections 6.3 and 6.4 can be skipped upon a first reading.

6.2 Reliability Analysis of PAGs

Assume that we are given a PAG tuple $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$. Intuitively, we are interested in finding out the probability that the intruder will reach a success state starting from one of the initial states. As shown above, in the absence of nondeterministic states we can compute this metric by using the steady state probabilities of the Markov chain. In the presence of nondeterministic states the intruder will choose transitions in order to maximize his probability of succeeding. For example, if an intruder reaches a nondeterministic state s with transitions to $s_1,...,s_k$, he will choose to transition to state $s_i (1 \le i \le n)$ which will maximize his probability of reaching a success state. This idea can be "formalized" using concepts from the theory of Markov Decision Processes [Alt99, Put94].

6.2.1 Value Iteration for PAGs

Given a state s, the set of successors of s is denoted by $succ(s)$. Formally, $succ(s)$ is equal to $\{s' \mid (s, s') \in \tau\}$. First, we define a *value function* $V : S \rightarrow R^+$. For all $s \in S_s$, $V(s) = 1.0$. For all states $s \in S \setminus S_s$ the value function is iterated according to the following equations until convergence.

$$V(s) = \begin{cases} max_{s' \in succ(s)} V(s') & \text{if } s \in S_n \setminus S_s \\ \sum_{s' succ(s)} P(s \rightarrow s') V(s') & \text{if } s \in S_q \setminus S_s \end{cases}$$

Let V^* be the value function after convergence. Intuitively, $\sum_{s \in S0} V^*(s)$ is the probability for the intruder to reach a success state if he "breaks" the nondeterminism to maximize the probability of succeeding. Therefore, the worst case reliability of the network is $1 - \sum_{s \in S0} V^*(s)$. This algorithm is known as *value iteration*. The justification of the value iteration algorithm as applied to PAGs is presented in Section 6.4.

6.2.2 Example Revisited

We implemented the value iteration algorithm in our attack graph post-processor and ran it on a slightly modified version of the intrusion detection example from Section 4. In the modified example, each attack has both detectable and stealthy variants. The intruder chooses which atomic attack to try next, and he has a certain probability of picking a stealthy or a detectable variant. We assigned imaginary probabilities of picking a stealthy attack variant as follows: 0.2 for sshd buffer overflow, 0.5 for ftp .rhosts, 0.05 for the other.

In this setup, the computed probability of intruder success is 0.2, and his best strategy is to attempt sshd buffer overflow on host ip_1, and then conduct the rest of the attack from that host. The only possibility of detection is the sshd buffer overflow attack itself, since the IDS does not see the activity between hosts ip_1 and ip_2. Given this context, a system administrator can answer the following question:

Question 4: The deployment of which security measure(s) will increase the likelihood of thwarting an attacker?

Answer: Installing an additional IDS component to monitor the network traffic between hosts ip_1 and ip_2 reduces the probability of the intruder remaining undetected to 0.025; installing a host-based IDS on host ip_2 reduces the probability to 0.16. Other things being equal, this is an indication that the former remedy is more effective.

6.3 Alternating Probabilistic Attack Graphs and Markov Decision Processes

In this section we show that probabilistic attack graphs can be reduced to Markov Decision Processes (without the reward function). We then demonstrate how we can assign a reward function to attack graphs such that standard MDP algorithms can be used to compute reliability metric of the network being modeled.

Definition 4 *[Alt99, Put94]* A Markov Decision Process is a tuple *(X, A, P, c)* where

- X is a finite state space. Generic notation for MDP states will be *x, y, z*.
- *A* is a finite set of actions. $A(x) \subseteq A$ denotes the actions that are available at state *x*. Set $K = (x, a) : x \in X, a \in A(x)$ is the set of state-action pairs. A generic notation for an action will be *a*.
- $P : X \times A \times X$ are the transition probabilities; thus, $P(xay)$ (also written as *Pxay*) is the probability of moving from state *x* to *y* if action *a* is chosen.
- $r : K \rightarrow R$ is an immediate reward. Cost may be equivalently viewed as a negative reward. We will freely use the term cost to mean negative reward, and vice versa.

An *execution fragment* (also known as history in the traditional MDP literature) of an MDP is a sequence $x_0a_1x_1 \ldots a_nx_n$ of alternating states and actions such that the sequence begins and ends with a state, and for all $0 < k \leq n$, $a_k \in A(x_{k-1})$ and $0 <$
$P(x_{k-1}, ak, x_k) \leq 1$. Given an execution fragment $e = x_0a_1x_1 \ldots a_nx_n$, the probability of the execution fragment (denoted by $P(e)$) is given by the following expression:

$$\prod_{K=1}^{n} P(x_{k-1}, a_k, x_k)$$

It is possible to convert a probabilistic attack graph into an MDP such that the behaviors of the PAG and the MDP are identical. To explain the conversion procedure, we define a restricted kind of probabilistic attack graph.

Definition 5 An *alternating probabilistic attack graph* or *APAG* is a tuple $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$, where S_n is a set of nondeterministic states, S_q is a set of probabilistic states, $s_e \in S_n$ is a nondeterministic escape state, $S = S_n \cup S_q$ is the set of all states, $\tau_n \subset S_n \times S_q$ is a set of nondeterministic transitions, $\tau_q \subseteq S_q \rightarrow S_n$ is a set of probabilistic transitions, $\pi : S_q \rightarrow S_n \rightarrow R$ are transition probabilities, $S_0 \subseteq S$ is a set of initial states, $S_s \subseteq S$ is a set of success states, and $L : S \rightarrow 2^{AP}$ is a labeling of states with a set of propositions true in that state.

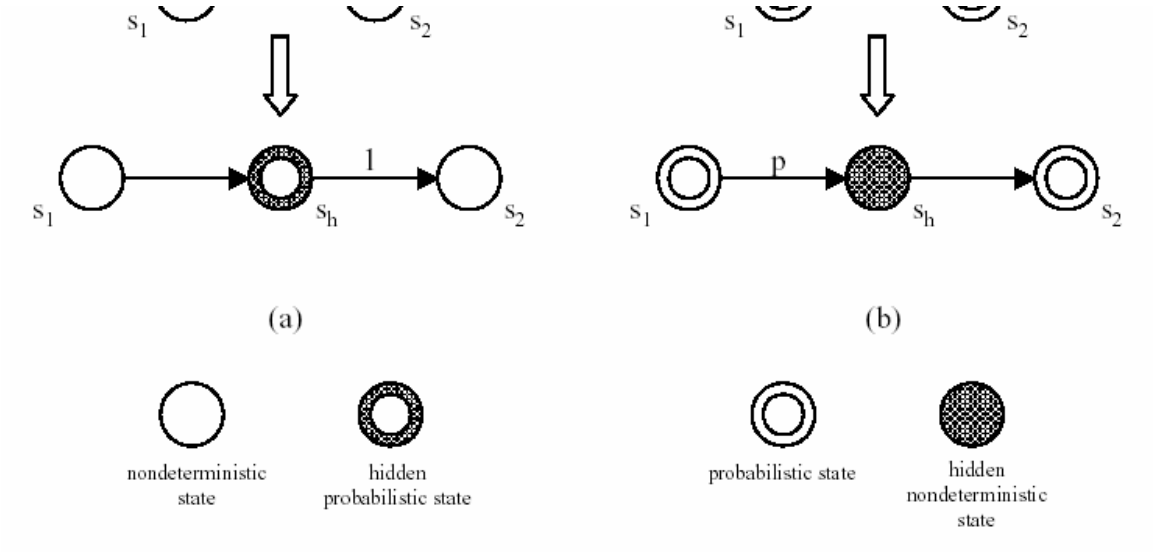


Figure 8: Converting PAG to APAG

An alternating probabilistic attack graph (APAG) *does not have any transitions between two nondeterministic or between two probabilistic states*. In other words, a nondeterministic state has transitions to probabilistic states only, and vice versa. An execution of an APAG will always have strictly alternating nondeterministic and probabilistic states.

Next we describe an algorithm that converts a PAG $G_p = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$ into an APAG $G_p^A = (S_n^A, S_q^A, s_e, S, \tau_n^A, \tau_q^A, \pi^A, S_0, S_s, L^A)$ that has equivalent behaviors. The algorithm works by adding *hidden* states and transitions to the graph such that every execution becomes strictly alternating, yet does not change its *observable* (non-hidden) components.

We start with $S_n^A = S_n$, $S_q^A = S_q$, $\tau_n^A := 0,, , \tau_q^A := 0$, $\pi^A := 0$, and $L^A = L$. Next,

1. Whenever τ has a transition from probabilistic state s_1 to nondeterministic state s_2, we add the transition to τ_q^A and its probability to π^A.

2. Whenever τ has a transition from nondeterministic state s_1 to probabilistic state s_2, we add the transition to τ_n^A.

3. Whenever τ has a transition between two nondeterministic states s_1 and s_2, we add a hidden probabilistic state s_h to S_q^A, an observable transition $s_1 \rightarrow s_h$ to τ_n^A, and a hidden transition $s_h \rightarrow s_2$ to τ_p^A, assigning the latter probability 1.0 in π^A (Figure8a). We also set $L^A(s_h) = L(s_1)$.

4. Whenever τ has a transition between two probabilistic states s_1 and s_2, we add a hidden nondeterministic state s_h to S_n^A, a hidden transition $s_h \rightarrow s_2$ to τ_n^A, and an observable transition $s_1 \rightarrow s_h$ to τ_p^A, assigning the latter the original probability p of going from s_1 to s_2 (Figure 8(b)). We also set $L^A(s_h) = L(s_1)$.

Let G_p be a PAG and G_p^A be the corresponding APAG. An execution fragment $e = s_0 s_1 \ldots s_n$ in G_p^A is called *proper* if the start and end states (s_0 and s_n) are observable states. Let e be a proper execution fragment of G_p^A. We define e^{obs} by removing hidden states and hidden transitions from e, i.e., restricting the execution to observable states and transition. Consider an execution fragment $e = s_0 s_1 \ldots s_n$. Let $S_P(e)$ be the set of probabilistic states in the set $(s_0, \ldots, s_{n-1})$. Define the probability of an execution fragment e (denoted by $P(e)$) as:

$$\prod_{s_i \in Sp(e)} P(s_i \rightarrow s_{i+1}).$$

In other words, the probability of an execution fragment is the product of the probabilities of the probabilistic transitions in it. The following lemma follows straight from the construction.

Lemma 7 Let G_p be a PAG and G_p^A be the corresponding APAG. Let e be a proper execution fragment of G_p^A. The following three statements are true:

1. e^{obs} is an execution fragment of G_p.

2. $P(e) = P(e^{obs})$, where the first probability is interpreted in G_p^A and the second probability is interpreted in G_p.

3. For all execution fragments e_1 of G_p there exists proper execution fragment e in G_p^A such that $e = e_1^{obs}$.

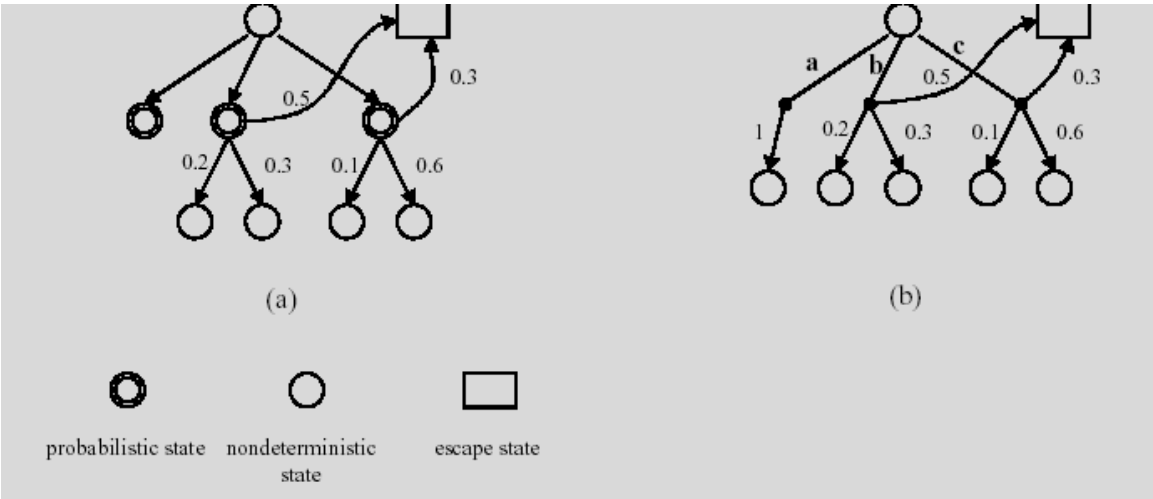


Figure 9: Converting an APAG to a MDP

Lemma 7 clearly shows that there is a one-to-one correspondence (given by *obs*) between proper execution fragments of a APAG and corresponding execution fragments of a PAG. Moreover, this correspondence preserves probabilities. We have shown that APAGs have the same expressive power as PAGs, so hereafter we consider them interchangeable.

An APAG $G= (S_n, S_q, s_e, S, \tau_n, \tau_q, \pi, S_0, S_s, L)$, has a direct interpretation as an MDP $M_G = (X, A, P, c)$, where $X = S_n$, $A = \tau_n$. That is, each action in the MDP represents a transition from a nondeterministic to a probabilistic state. Further, let $x, y \in X$ and $a \in A(x)$, so that a represents a transition from x to some probabilistic state s_q in the APAG. Then we have $P(x, a, y) = \pi(s_q)(y)$.

It is preferable to have all APAG success states represented explicitly as MDP states, so that we can reason about attacks in the MDP context. For this reason, we add a hidden nondeterministic state (and a transition thereto) to every probabilistic success state in the APAG. We omit proofs of equivalence of an APAG before and after this modification.

Figure 9(a) shows an example APAG, with the corresponding MDP shown in Figure 9(b). The nondeterministic transitions from the root node in the APAG are represented by the MDP actions **a**, **b**, and **c**. The leftmost leaf in the APAG is a probabilistic success state; in the MDP it is represented by the appended hidden nondeterministic state.

This however, plays a role in our interpretation of results obtained through MDP algorithms. Finally, we can choose the reward function r depending on the questions we are trying to answer.

Let $e = s_0^n s_1^p s_1^n \ldots s_1^{n-1} s_n^p s_n^n$ be an execution fragment of the APAG G, where s_k^n and s_k^p represent nondeterministic and probabilistic states respectively. Let $mdp(e) = e^{mdp} = s_0^n t_1^n s_1^n \ldots s_n^n$, where t_i^n is the action that corresponds to the transition $s_{i-1}^n \rightarrow s_i^p$. Notice that in *mdp(e)* probabilistic states do not occur. The proof of the following lemma follows straight from the construction.

Lemma 8 Let G be a APAG and M_G be the corresponding MDP. Let e be an execution fragment of G and *mdp(e)* be the corresponding execution fragment in the MDP M_G. The following statements are true.

1. *mdp(e)* is an execution fragment of the MDP M_G.

2. $P(e) = P(mdp(e))$, where *P(e) and P(mdp(e))* are interpreted in G and M_G respectively.

3. For all execution fragments e_m in the MDP M_G, there exists an execution fragment e in G such that $mdp(e) = e_m$.

6.4 Correctness of the Value Iteration Algorithm for Attack Graphs

Let $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$ be a PAG, and $G^A = (S_n^A, S_q^A, s_e, S^A, \tau_n^A, \tau_q^A, \pi^A, S_0, S_s, L^A)$ be the corresponding APAG. Recall that the APAG G^A is obtained from the PAG G by adding hidden states whenever there is a transition between two nondeterministic or probabilistic states (see Section 6.3). An APAG $G = (S_n, S_q, s_e, S, \tau_n, \tau_p, \pi, S_0, S_s, L)$ has a direct interpretation as an MDP $M_G = (X, A, P, r)$, where $X = S_n$, $A = \tau_n$. That is, each action in the MDP represents a transition from a nondeterministic to a probabilistic state. Further, let $x, y \in X$ and $a \in A(x)$, so that a represents a transition from x to some probabilistic state s_q in the APAG. Then we have $P(x, a, y) = \pi(s_q)(y)$. We first demonstrate that the *value iteration algorithm* (or *VI* for short) on the APAG G^A is simply a transformed version of the value iteration algorithm on the corresponding MDP M_G with an appropriate reward function r. After that, we prove that the value iteration algorithm on the PAG and the corresponding APAG converge to the same value. The advantage of this approach is that all the technical results in the context of value iteration in MDPs can be directly applied to value iteration in PAGs [Put94, Chapter 9].

6.4.1 Correspondence Between Value Iteration in MDPs and APAGs

Consider a MDP $M = (X, A, P, r)$. A *value function* is positive real valued function $V : X \rightarrow R^+$. The value iteration algorithm uses the following equation to update the function V:

$$V(x) = \max_{a \in A(x)} [r(x, a) + \sum_{y \in X} P(x, a, y) V(y)]$$

Technical conditions that guarantee the convergence of the value iteration algorithm can be found in [Put94, Chapter 9].

Let G^A be an APAG and M_G be the corresponding MDP. Recall that we assumed that all success states in G^A are nondeterministic states so that they are explicitly represented in the MDP M_G. Before we proceed, we need to slightly modify the MDP M_G. We add a new state s_{new} and action a_{new} to the MDP M_G. The only action allowed from s_{new} is a_{new} ($A(s_{new}) = (a_{new})$ and $P(s_{new}, a_{new}, s_{new}) = 1.0$ (so by definition $P(s_{new}, a_{new}, s) = 0.0$ if $s \neq s_{new}$). Moreover, we add the action a_{new} to the action set corresponding to the success states S_f and for all $s \in S_f$ we have $P(s, a_{new}, s_{new}) = 1.0$ (so by definition $P(s, a_{new}, s') = 0.0$ if $s' \neq s_{new}$). We have the following reward function r:

$$R(s, a) = \begin{cases} 1.0 \text{ if } s \in S_s \text{ and } a = a_{new} \\ 0.0 \text{ otherwise} \end{cases}$$

We have a value function that assigns 1.0 to every state. It is easy to see that the value function assigns 1.0 to the newly added state s_{new} and 1.0 to a state in the set S_f. For states that are not in the set $\{s_{new}\} \cup S_s$ the value function V changes according to the following equation:

$$V(x) = \max_{a \in A(x)} \sum_{y \in X} P(x, a, y)V(y)$$

$$= \max_{s_q succ(x)} \sum_{y \in X} P(s_q \rightarrow y)V(y)$$

The second equation follows from the construction of the MDP M_G from the APAG G^A. Recall that actions in the MDP correspond to the transitions from nondeterministic to probabilistic states. Next we extend the value function V to probabilistic states S_q by defining $V(s)$ (for all $s \in S_q$) as:

$$\sum_{y \in X} P(s_q \rightarrow y)V(y)$$

Notice that in an APAG only successors of a probabilistic state s are nondeterministic state, so $V(y)$ is well defined. Using this definition the value iteration algorithm can be re-written as:

$$V(s) = \begin{cases} \max_{s' \in succ(s)} V(s') & \text{if } s \in S_n \setminus S_s \\ \sum_{s' \in succ(s)} P(s \rightarrow s')V(s') & \text{if } s \in S_q \setminus S_s \end{cases}$$

The value iteration (VI) equation given above was obtained by transforming the VI equation for the corresponding MDP. Moreover, the equation we obtain is exactly the VI equation for an APAG that was provided earlier (see Section 6.2).

6.4.2 Correspondence Between Value Iteration in MDPs and PAGs

Let $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$ be a PAG, and $G^A = (S_n^A, S_q^A, s_e, S^A, \tau_n^A, \tau_q^A, \pi^A, S_0, S_s, L^A)$ be the corresponding APAG. Recall that G^A is obtained from G by adding hidden states whenever there is a transition between two nondeterministic or probabilistic states (see Figure 8). Suppose there is a transition between two nondeterministic states s_1 and s_2 in G. In G_A, we add a new probabilistic state s_h and add transitions $s_1 \rightarrow s_h$ and $s_h \rightarrow s_2$, where the probability of the transition $s_h \rightarrow s_2$ is 1.0. Consider the i-th iteration of the VI algorithm in G. In this case, the value $V(s_2)$ in the $(i-1)$-the iteration is used to update the value of the state s_1. Now consider the value iteration algorithm in G^A. The value $V(s_h)$ of the hidden state s_h in the $(i-1)$-th iteration is used to update the value of $V(s_1)$ in the i-th iteration. It is easy to see that $V(s_h)$ in the $(i-1)$-th iteration is $V(s_2)$ in the $(i-2)$-th iteration. Therefore, hidden states *add a delay of* 1 *in the value iteration algorithm*. The case for transition between two probabilistic states is analogous.

Consider a PAG $G = (S_n, S_q, s_e, S, \tau, \pi, S_0, S_s, L)$. The equation for the value iteration algorithm without delay is:

$$V^i(s) = \begin{cases} 1.0 & \text{if } s \in S_s \\ \max_{s' \in succ(s)} V^{i-1}(s') & \text{if } s \in Sn \setminus Ss \\ \sum_{s' \in succ(s)} P(s \rightarrow s')V^{i-1}(s') & \text{if } s \in S_q \setminus S_s \end{cases}$$

We have added the iteration index i to the VI algorithm so that we can refer to it in the proof. The value iteration algorithm with the delay is:

$$V_1^i(s) = \begin{cases} 1.0 & \text{if } s \in S_s \\ \text{Max}\{\max_{s' \in succ(s) \cap Sn} V^{i-2}(s'), \text{Max}\{\max_{s' \in succ(s) \cap Sq} V^{i-1}(s') & \text{if } s \in Sn \setminus Ss \\ \sum_{s' \in succ(s) \cap Sq} P(s \rightarrow s')V^{i-2}(s') + \sum_{s' \in succ(s) \cap Sn} P(s \rightarrow s')V^{i-1}(s') & \text{if } s \in S_q \setminus S_s \end{cases}$$

Initially, both sequences start with the value functions V^0 and V_1^0 that assign 1.0 to states in S_f and 0.0 to all other states. Notice that in the value iteration algorithm for V_1^i there is delay of 1 added (the $(i-2)$-th value for all $s \in S$, $V(s') \in$

$V(s)$ and $V^i(s') \in V^i(s)$. For $i \geq 2$i, the following inequality also holds for all $s \in S$ and $i \geq 2$:

$$V^i(s) \geq V_1^i(s) \geq V^{i-2}(s)$$

The equation given above directly follows from the monotonicity property and the equations that define value iteration.

Suppose V converges to V_* pointwise, i.e., for all $s \in S$, $V(s) \rightarrow V_*(s)$. Next we prove that for all $s \in S$, if $V_1^i(s) \rightarrow V_*(s)$, then $V_1^i(s) \rightarrow V_*(s)$. This proves that V_1also converges to V_*. By definition of convergence, for all $e > 0$, there exists a positive integer $N(e)$ such that for all $i > N(e)$ we have:

$$\left| V_*(s) - V^i(s) \right| < e$$

Assume that we are given a $B > 0$. It is easy to see that the limit $V_*(s) \geq V^i(s)$ for all i (this follows from the fact that $V^i(s)$ is a monotonic sequence). Therefore, we have the following inequality:

$$\left| V_*(s) - V_1^i(s) \right| \leq \left| V_*(s) - V^{i-2}(s) \right|$$

The equation given above follows from the inequality $V_1^i(s) \geq V^{i-2}(s)$ for all s. Since $V^i(s) \rightarrow V_*(s)$, there exists an $N(B)$ such that if $i > N(B)$, then:

$$\left| V_*(s) - V^i(s) \right| < B$$

By the argument given above $\left| V_*(s) - V_1^i(s) \right| < B$ for $i > N(B) + 2$. This proves that $V_1^i(s) \rightarrow V_*(s)$. Conversely assume that V_1 converges to V_*'. Using the inequality given below it is easy to prove that $V^i(s) \rightarrow V_*'(s)$.

$$\left| V_*'(s) - V^i(s) \right| < \left| V_*'(s) - V_1^i(s) \right|$$

Therefore, we prove that the value iteration algorithm with and without delay converge to the same value. The VI algorithm with delay is essentially the VI algorithm on the APAG G^A, which was derived from the VI algorithm on the corresponding MDP. Therefore, the correctness of the VI algorithm on the PAG G follows.

7 Summary of Contributions and Future Work

Our foremost contribution is the automatic generation of attack graphs. Our key insight is that an attack is equivalent to a counterexample produced by off-the-shelf model checkers; the attack/counterexample is a witness to a violation of a safety property. By a small, but critical enhancement to an existing model checker, i.e., NuSMV, we can easily produce attack graphs automatically; moreover, these graphs are succinct and exhaustive. A by-product of this part of our work is showing, by example, what level of abstraction is appropriate for modeling attacks. We use simple state machine specifications to model not just intruder behavior (by a set of atomic attacks), but also normal system behavior, system administrator recovery actions, and connectivity (communication) between subsystems.

Our second most important contribution is support for a range of formal analyses of attack graphs. Security analysts use attack graphs informally for attack detection, defense, and forensics. In this paper, we explain how they can now use our minimization analysis technique on attack graphs to more precisely answer questions like “Which security measure should I deploy in order to thwart this set of attacks?” and “Which set of security measures should I deploy to guarantee the safety of my system?” To do reliability analysis, we annotate attack graphs with probabilities and then interpret them as Markov Decision Processes (MDP). Then, by using MDP algorithms such as value iteration, security analysts can more precisely answer questions like “Will deploying this intrusion detection system increase or decrease the likelihood of thwarting this type of attack?”

On the theoretical front, we have so far restricted our work to only safety (invariant) properties. To exploit the full power of model checking, we need a method of generating attack graphs for more general classes.

AG(server.user.request → **AF**(server.user.acesss))

This property would not be true if the server can be disabled using a denial-of-service attack. Another such liveness property is that a legitimate user's transaction will finish despite intruder interference. We plan to explore generation of attack graphs for universally quantified fragments of Computational Tree Logic and Linear Temporal Logic. On the practical front, we plan to conduct larger case studies to illustrate the usefulness of automatically generating attack graphs. To make our tool suite more usable by security experts and system administrators, we see the value of building a library of specifications of atomic attacks. Our hope is that increasing this arsenal of specifications outpaces the growth in the arsenal of known attacks; we can potentially discover new, unexpected attacks, and hence identify new network vulnerabilities. Finally, we also intend to build a tool that merges our work on attack graphs with existing intrusion detection technologies. The tool is intended help security analysts evaluate and enhance the security of a network.

References

[ADP80] G. Ausiello, A. D'Atri, and M. Protasi. Structure preserving reductions among convex optimization problems. *Journal of Computational System Sciences*, 21:136–153, 1980.

[ALT99] Eitan Altman. *Constrained Markov Decision Processes*. Chapman & Hall/CRC, 1999.

[BRY86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CGP00] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[CLR85] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. MIT Press, 1985.

[DAC94] M. Dacier. *Towards Quantitative Evaluation of Computer Security*. PhD thesis, Institut National Polytechnique de Toulouse, December 1994.

[DUR95] Richard Durrett. *Probability: Theory and Examples*. Duxbury Press, 1995. 2nd edition.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP Completeness*.
W.H. Freeman and Company, San Francisco, 1979.

[JW01] Somesh Jha and Jeannette M.Wing. Survivability analysis of networked systems. In *Proceedings of the International Conference on Software Engineering*, May 2001.

[NUS] NuSMV. Nusmv: a new symbolic model checker. http://afrodite.itc.it:1024/ nusmv/.

[ODK99] R. Ortalo, Y. Dewarte, and M. Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering*, 25(5):633–650, September/October 1999.

[PS98] C.A. Phillips and L.P. Swiler. A graph-based system for network vulnerability analysis. In *New Security Paradigms Workshop*, pages 71–79, 1998.

[PUT94] M. Puterman. *Markov Decision Processes*. John Wiley & Sons, New York, NY, 1994.

[RA01] R.W. Ritchey and P. Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 156–165, May 2001.

[SMV] SMV. Smv: a symbolic model checker. http://www.cs.cmu.edu/ modelcheck/.

[SPEC00] L.P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *Proceedings of the DARPA Information Survivability Conference and Exposition*, June 2000.

[STE] Peter Stephenson. Using formal methods for forensic analysis of intrusion events - a preliminary examination. White Paper, available at http://www.imfgroup.com/Document Library.html.

[VS01] Alfonso Valdes and Keith Skinner. Probabilistic alert detection. In *Proceedings of Recent Advances in Intrusion Detection (RAID)*, 2001.

Survivability Analysis of Networked Systems

S. Jha[1] J. Wing[2]

October 2000
CMU-CS-00-168

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This paper was submitted to the International Conference on Software Engineering 2001, Toronto, May 12-19, 2001.

1) Computer Sciences Department, University of Wisconsin, Madison, WI 53706. 2) Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213.

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, F33615-93-1-1330, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031 and in part by the National Science Foundation under Grant No. CCR-9523972. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory or the U.S. Government.

Abstract

Survivability is the ability of a system to continue operating despite the presence of abnormal events such as failures and intrusions. Ensuring system survivability has increased in importance as critical infrastructures have become heavily dependent on computers. In this paper we present a systematic method for performing survivability analysis of networked systems. An architect injects failure and intrusion events into a system model and then visualizes the effects of the injected events in the form of ***scenario graphs****. Our method enables further global analyses, such as reliability, latency, and cost-benefit analyses, where mathematical techniques used in different domains are combined in a systematic manner. We illustrate our ideas on an abstract model of the United States Payment System.*

Keywords: survivability, model checking, reliability analysis, cost analysis, Markov Decision Processes, fault-tolerance, security

1 Introduction and Motivation

Increasingly our critical infrastructures are becoming heavily dependent on computers. We see examples of such infrastructures in all domains, including medical, power, telecommunications, and finance. Whereas automation provides society with the advantages of efficient communication and information sharing, the pervasive, continuous use of computers exposes our critical infrastructures to a wider variety and higher likelihood of accidental failures and malicious attacks. Disruption of services caused by such undesired events can have catastrophic effects, including loss of human life.

Survivability is the ability of a system to continue operating in the presence of accidental failures or malicious attacks [7]. We use the term *fault* for both accidental failures (e.g., a disk crash) and malicious attacks (e.g., a denial-ofservice attack). The precise semantics of *continuous operation is* application dependent; it is related to critical services that the system provides. For example, check clearing is a critical service of a banking system, and a survivable banking system will continue providing this service despite the presence of faults.

In this paper we present a method for analyzing a networked system for survivability. *A networked system* consists of nodes and links connecting the nodes. Communication between the nodes occurs by passing messages over the links. An *event* in the system can be either a user event (e.g., a user issues a check), an internal event (e.g., a user's account is debited), a communication event (e.g., sending a message between two banks), or a fault (e.g., a bank under a malicious attack). *A service is* associated with a *start event (e.g.,* a user issues a check) and an *end event (e.g.,* the check clears). The start event and the end event correspond respectively to when "a service is issued" and when a "service is finished."

Our main goal is to provide information to the system architect during the design phase, the early planning stage of the software lifecycle. With this information, the architect can weigh the pros and cons of decisions related to survivability. The method we present in this paper, however, is just as suitable for post facto analysis of existing systems.

Our method is general enough to support many different types of analysis. In this paper we focus on three specific kinds of questions.

Question 1: *What is the effect of a fault?*

Example: Imagine an architect is designing a power grid. He wants to know the effect of an outage of a power plant located in upstate New York on customers living hundreds of miles away in western Pennsylvania.

Answer (Fault-Effect Analysis): Using our method the architect can visualize the global effect of a local fault through a data structure that we call a *scenario graph.* In our method, we automatically generate scenario graphs using model checking.

Question 2: *What is the reliability and latency of a service?* Here, reliability is defined as the probability that a service that has been issued will finish. Latency measures the expected time it takes a service to finish.

Example: Suppose an architect designing a banking system wants to find out the probability that a check issued actually clears.

Answer (Reliability and Latency Analysis): To find the reliability of the banking system with respect to the check clearing service, we query an annotated scenario graph. The architect first identifies a set of "critical" elements in the network, i.e., nodes and links whose failures would have a severe effect on the provision of the service in question. He then assigns probabilities to each fault (i.e., the failure of each node or link). Then, using our method, he can automatically compute both the reliability and latency of the network.

Question 3: *Given cost constraints, which network nodes/links should be upgraded to maximize benefit (e.g., reliability)?*

Example: Suppose an architect is allowed to spend newly allocated funds to upgrade a fraction of the network's links to newer links that are faster and more reliable. Given the constraints imposed by his manager's limited budget, which links should he choose to upgrade to maximize the network's reliability?

Answer (Cost-Benefit Analysis): To perform a cost-benefit analysis, we further extend our annotated scenario graphs with additional cost information related to upgrading the links. We then can automatically compute how to maximize a given benefit given a set of cost constraints.

Survivability analysis is fundamentally different from analysis of properties found in other areas (e.g., algorithm analysis of fault-tolerant distributed systems, reliability analysis of hardware systems, and "security" analysis of computer systems). First, survivability analysis must handle a *broader range of faults* than any of these other areas; we must minimally handle both accidental failures and malicious attacks. To achieve this goal our method allows an architect to incorporate any arbitrary type of fault in the system model; however, we still allow distinctions among faults by assigning different weights (e.g., probability of occurring, cost to repair, etc.) to each fault.

Second, *events may be dependent on each other,* especially fault events. In contrast, for ease of analysis, most work in the fault-tolerant literature makes the *independence assumption:* assume that abnormal events are independent. We cannot make this assumption in analyzing systems for survivability. For example, if a server crashes, then it is easier for a malicious intruder to spoof the crashed server; the chance that an intruder will succeed in spoofing a server depends on the event that the server crashes. Or, if an attacker learns how to compromise one disk of a replicated server, then he can easily compromise the replicas too; the chance of bringing down an entire service depends on the likelihood of success of the original attack. In our method we allow users to express such dependencies. Representing dependence between events allows us to model phenomena such as *correlated attacks,* where local attacks might not succeed, but when they occur in tandem or in succession they can have a severe effect on the system. Distributed denial-of-service attacks is an example of a correlated attack (see CERT advisory CA-2000-0). Representing dependence also allows us to handle *cascading effects,* where one fault triggers another, which then triggers another, and so on. While it is cleaner to design a system to avoid cascading effects (e.g., by using a strict locking protocol to avoid cascading aborts in a transactional database), in practice it may be impossible to anticipate faults induced by a system's environment that violates the assumptions made by the system's original designer. Since survivability is of particular concern to those building systems of systems, system architects will have to face the possibility of cascading effects in their analysis.

Third, survivability analysis should also be *service dependent.* For example, the architect for a banking system might choose to focus on the check clearing service as being critical, although the banking system provides other services such as accounting, auditing, and cash distribution; for a different analysis, cash distribution might be the critical service to focus on. Taking into consideration the specific service a system is to provide enables more targeted analysis, which is often amenable to fully automated support. Also a method that focuses the architect's attention on specific services rather than the general system design is likely to be more appreciated and better understood by the end customer (who cares about the reliability of the applications' services). The analyses in our method are all driven by the properties that the architect specifies as they relate to a critical service.

Finally, survivability analysis deals with *multiple dimensions.* It simultaneously deals with functional correctness (modeling the service itself), fault-tolerance (modeling the effects of accidental failures), security (modeling the effects of malicious attacks), reliability (the likelihood of a service finishing), performance (network latency), and cost. To achieve this goal, the analytical approach described in this paper combines several different kind of analysis techniques into one framework.

The next section introduces *constrained Markov Decision Processes* which form the basis for reliability, latency, and cost-benefit analysis. A general overview of our method appears in Section 3. We describe a small example based on the United States Payment System in Section 4, which we use as a running example throughout the remainder of the paper. Section 5 provides additional details

related to each step in our method. Section 6 briefly describes a prototype tool *Trishul* that we have implemented based on our method, and briefly describes two case studies that we have performed. Sections 7 and 8 discuss related work and conclusions respectively.

2 Model of Computation

Our formal model is based on *constrained Markov Decision Processes* or simply CMDPs. CMPDs are a generalization of Markov chains, where the transition probabilities depend on the past history. CMDPs enable us to model history dependent transition probabilities and provide a framework to perform cost-benefit analysis. Our exposition of CMDPs is based on Altman [2]. A CMDP is 5-tuple *(S, A, P, c, d)* where:

- S is a finite state space.
- A is a finite set of actions. For a state $s \in S$, $A(s) \subseteq A$ is the set of actions available at state s.
- P are transition probabilities, where $P_{sas'}$ is the probability of moving from state s to s' if action a is chosen.
- $c : (S \times A) \rightarrow R$ is the immediate cost, i.e., $c(s, a)$ denotes the cost of choosing action a at state s. This cost will be related to the value function to be minimized.
- $d : (S \times A) \rightarrow R^k$ is a k-dimensional vector of immediate costs. This will be related to cost constraints.

A Markov Decision Process (MDP) *is* a CMDP without the last component d.

History at time t (denoted by h_t) is the sequence of states encountered and actions taken up to time t. *A policy* u takes into account the history h_t and determines the next action at time t. Specifically, $u_t(a \mid h_t)$ *is* the probability of taking action a given history h_t. A policy u defines *a value function* $V^u : S \rightarrow R$, where $V^u(s)$ is the expected cost of the actions taken if the CMPD uses policy u and starts in state s (the cost c is used to define expected cost). The technical definition of V^u can be found in [2]. Analogously, starting in state s let the expected value of the immediate costs d under the policy u be denoted *by* $D^u(s)$. Since the result of d *is* a k-dimensional vector, $D^u(s)$ is also a k-dimensional vector of real numbers. Assume that we are also given a k-dimensional vector $C = (c1, \ldots, ck)$, where ci is the cost constraint on the i-th component of $D^u(s)$. Our aim is to find a policy that minimizes the value function V^u given the constraint imposed by the vector C, or

Given an initial state $s_o \in S$, find a policy u that minimizes $V^u(so)$ subject to $D^u(s_o) < C$.

Remark: Do not confuse a Markov process with a Markov policy, which is a policy where the probability of an action depends only on the current state of the CMDP and not the entire history.

Example 2.1 Imagine a bakery where there can be at most 10 customers waiting at any time. At each time the bakery manager has the option of having one or two servers behind the counter. The state of the CMDP corresponds to the number of servers behind the counter and the number of customers waiting. The action at each state is to decide on how many servers should be behind the counter. In Figure 1 we show a few transitions. Consider the transition from state (S=1, C=m) to (S=2, C=m-1). The action label $a = 2$ on the transition indicates that the manager decided to switch to two servers behind the counter. The probability that a waiting customer leaves with his/her order is 0.5 or 0.75 depending on whether there are one or two servers behind the counter. Notice that the probability that a customer gets serviced is higher when there are two servers behind the counter. Therefore, the transition from state (S=1, C=m) to (S=2, C=m-1) has probability 0.75. The rest of the transitions have a similar explanation. Given a state and an action, the probability that a customer is serviced in the next time period determines the cost function c. For example, the cost of the state action pair h (S=1, C=m), a=1 i is -0.5 because if an action a=1 is chosen from the state the expected number of customers that are serviced during the next time step is 0.5. Notice that the negative of the cost determines the throughput, i.e., the expected number of

customers that are serviced in the next time period. The number of servers behind the counter determines the cost function d, i.e., two servers cost more than one. The aim of the manager is to maximize expected throughput (or minimize expected cost related to c) given a constraint on the wages of the servers. Achieving this goal can be easily seen as a problem of value maximization under cost constraints and naturally fits the CMDP framework. The optimal policy for this CMDP will indicate to the bakery manager when to change the number of servers behind the counter.

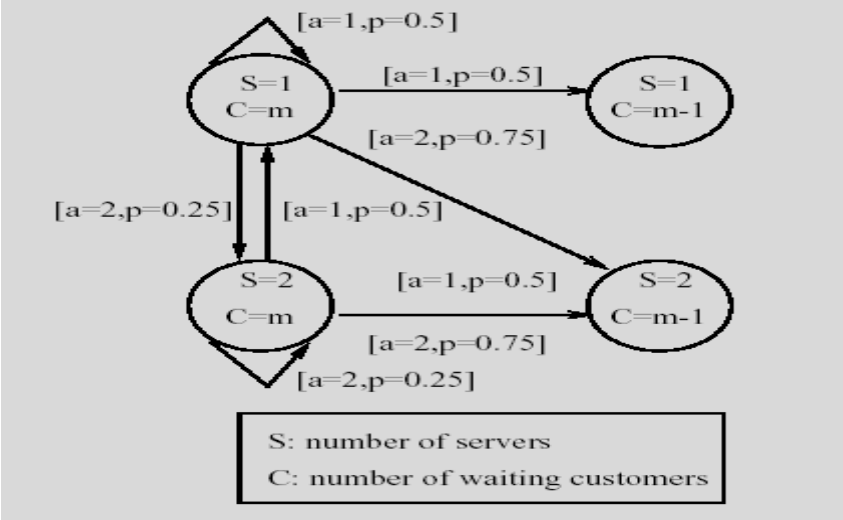


Figure 1: A Bakery

3 The General Method

In this section we provide a brief overview of our method; Section 5 gives more details about the techniques we use and our implementation. In steps 1, 2, and 3 we model the network, inject faults into our model, and specify survivability related properties. Then in steps 4, 5, and 6 we analyze the effects of faults, perform reliability and latency analysis, and do cost-benefit analysis to parallel answering the three kinds of questions posed in the introduction.

3.1 Step 1: Model the Network

First, the architect models a networked system, which can be done using one of many formalisms. We choose to use *state machines* and we use them to model both network nodes and links. We use shared variables to represent communication between the state machines.

3.2 Step 2: Inject Faults

Both links and nodes may be faulty. With our state machine model of the networked system, we need not make a distinction between nodes and links when considering faults. That is, a link is simply a node that passes data between two other nodes. Injecting a fault then requires first representing that a fault has occurred and then determining the behavior of the faulty node for each kind of fault that may occur. The exact behavior of a faulty node, specified by the architect, depends on the application.

To represent faults in our method, for each state machine representing a node, we introduce a special variable called **fault,** which can range over a userspecified set of symbolic values. For example, the following declaration states that there are three modes of operation for a node, representing whether it

is in the normal mode of operation, failed, or compromised by an intruder.

fault: { normal, failed, intruded }

Given this simple representation, we can then choose to specify the precise behavior of the node in each mode of operation. For example, for any given state we can specify that the machine makes a transition from the normal mode of operation to one of the abnormal modes **(failed** or **intruded)** and further specify what state the machine is in once such a transition occurs. We also have the option of leaving state transitions completely nondeterministic.

3.3 Step 3: Specify Survivability Properties

The architect specifies properties related to survivability using some kind of formal logic. In our method, we use a temporal logic called *Computation Tree Logic* (CTL), but other temporal logics such as *Linear Time Logic* [15] would also be appropriate.

In this paper, we focus on two classes of survivability properties: *fault* and *service* related. The first class captures properties of the networked system under scrutiny when it enters a faulty state. The second class captures properties specific to the system's services.

3.4 Step 4: Generate Scenario Graphs

Given a state machine model, *M,* of the networked system (with injected faults) and a survivability property, *P*, we then generate *a scenario graph,* which is a concise representation of a set of traces of *M* with respect to *P*. For fault properties, *a fault scenario graph* represents all system traces that end in a faulty state; for service properties, *a service success (fail) scenario graph* represents all system traces in which an issued service successfully finishes (fails to finish). An architect can use scenario graphs to visualize the effects of injected faults on a certain service. (In the operational security literature, scenario graphs are similar to *attack state graphs [13].)*

3.5 Step 5: Reliability and Latency Analysis

Once we have a scenario graph, we can perform further analyses, such as reliability and latency analysis. First, the architect specifies the probabilities of certain events of interest, such as faults, in the system. Since we do not assume independence of events, we use a formalism based on *Bayesian networks [14]* to specify the conditional probabilities of the events. We combine the specified probabilities with the scenario graph to obtain an MDP. We can then readily compute reliability and latency by solving for optimal policies using the relevant cost functions c, i.e., for reliability analysis the cost function is identically zero; for latency analysis, it is a function of the times associated with making state transitions.

An advantage of our method is that an architect need not specify probabilities for all events; an MDP can have both probabilistic and nondeterministic transitions.

3.6 Step 6: Cost-Benefit Analysis

In this step we transform the MDP from Step 5 into a CMDP. First we enhance the MDP's set of actions A with actions corresponding to decisions that an architect has to make. For example, these additional actions might correspond to upgrading links to produce a more reliable/faster system, and the architect must decide which links to upgrade. Each added action has a cost; the architect wants to simultaneously minimize cost and maximize some benefit (e.g., reliability). Thus, we also associate costs with these actions and provide constraints on these costs (i.e., specify the function *d* in the definition of CMDPs). The optimal policy corresponding to the CMDP so constructed provides the architect with the optimal decision under the specified cost constraints.

4 Example

We consider a simplified model of the United States Payment System, depicted in Figure 2. There are three levels of institutions: *Federal Reserve Banks* at the top, *money centers* in the middle, and small *banks* at the bottom. If banks are connected to the same money center, then transactions between them are handled by the money center; there is no need to go through the Federal Reserve Banks. For a detailed description of the system see [11].

To illustrate the architecture, suppose a customer A writes a 850 check to customer C so that the check has a source address Bank-A and destination address Bank-C. The following steps occur for the issued check to clear:

1. Bank-A and Bank-C are not connected through a money center, so the check is then sent to a money center connected to Bank-A. In this case, let's choose money center MC-1.
2. The check is then transferred to the Federal Reserve Bank closest to MC-1, in this case FRB-2.
3. The check is then transferred to the Federal Reserve Bank that has jurisdiction over Bank-C, in this case FRB-3.
4. The check finally makes it way to Bank-C through the money center MC-3.

In Figure 2 the path of the check is shown using dot-dashed lines.

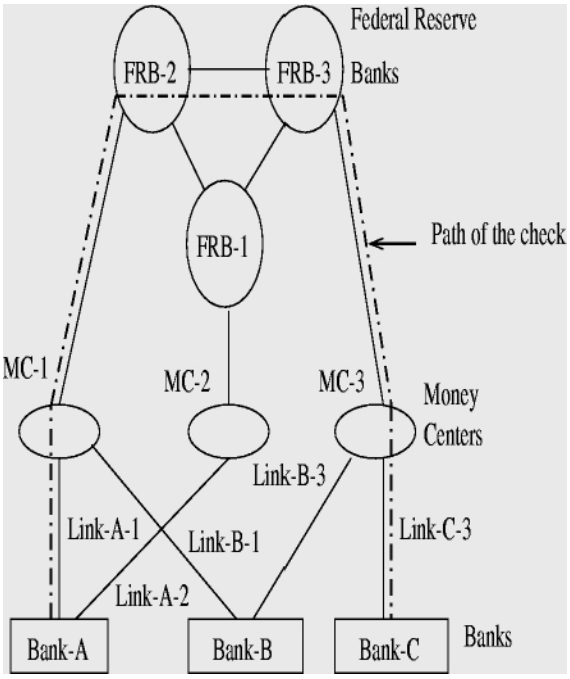


Figure 2: United States Payment System

5 Detailed Description

We now present the details of each step in our method in more detail, illustrating them with the check clearing example.

5.1 Step 1: Model the Network

We model each node and link in the system as a finite state machine, and the entire networked system as the composition of these machines. In our implementation, we use the model

checker **NuSMV [1],** and hence we use **NuSMV's** input language to describe the state machines representing a given system. Using this off-the-shelf model checker makes it convenient for us at later steps in our method to perform further global analyses; **NuSMV's** output lets us automatically derive information that we would otherwise have to reconstruct.

In our banking example, we use state machines to model the banks, the money centers, the Federal Reserve Banks, and the links. Each element in the banking infrastructure corresponds to a **MODULE** description in **NuSMV** and communication is achieved by parameter passing. We make some simplifying assumptions in the model of our system: (1) There is just one user who issues checks; the source and destination address of these checks are decided nondeterministically, i.e., the source address can be banks A, B, or C, and similarly for the destination; (2) There is only one check active at any time, and the exact amount of the check is irrelevant.

5.2 Step 2: Inject Faults

Next we inject faults in our model by including a special state variable **(fault)** with each state machine to indicate the mode of operation. We modify the specification of each state machine to take into consideration its faulty modes of operation.

In our banking example, what faults we inject and how we handle them in our model are based on the following assumptions:

• The only network elements that can be faulty are (1) links between the banks and the money centers; and (2) small banks, representing that penetration by a malicious intruder has occurred (i.e., **fault** = **intruded).** No other links or institutions may become faulty and banks cannot fail accidentally.
• When a link is faulty, it blocks all messages and consequently no message ever reaches the recipient.
• Links may become faulty at any time. Thus, in our finite state machine model of a link, we allow a nondeterministic transition to the state where **fault** is equal to **failed.** The third value **intruded** for the variable **fault** is not used in this case.
• Banks can sense a faulty link and route the checks accordingly.

These assumptions show how we take into consideration the semantics of the application; e.g., we are implicitly assuming that Federal Reserve Banks are impenetrable and links between them are highly reliable and secure.

Our model reflects the following behavior. Under the normal mode of operation, a bank receives a check (nondeterministically issued by the user~ with its source address. Depending on the destination address of the issued check, the bank either clears it locally or routes it to the appropriate money center. For example, if a check with source address A and destination address B is issued, then it is sent to the money center MC-1 and then sent to bank B. On the other hand, a check with source address A and destination address C has to clear through the Federal Reserve Banks (as in Figure 2). If a bank is faulty, then checks are routed arbitrarily by the intruder (thereby ignoring the check's destination address). A bank can then at any time nondeterministically transition from the normal mode **(fault=normal)** to the intruded mode **(fault=intruded)**. Once the bank is faulty it stays in that state forever.

The precise behavior of a faulty node depends on the application, but two types of behaviors under failure conditions are common. In the case of a stuck-at fault the node becomes stuck, i.e., it accepts no input on its channel and consequently produces no output. A node with a Byzantine fault exhibits completely nondeterministic behavior, i.e., accepts any inputs and produces arbitrary outputs. A Byzantine fault can also be used to model an intruded node.

5.3 Step 3: Specify Survivability Properties

In this step, we specify survivability properties in CTL, a logic chosen for convenience since the model checker we use accepts CTL specifications. Although CTL is a rich logic and allows us to express a variety of properties, we focus on two classes of survivability properties: fault and service related.

Fault Related Properties

Suppose we want to express the property that it is not possible for a node *N* to reach a certain unsafe state if the network starts from one of the initial states. The precise semantics of an unsafe state depends on the application. Let the atomic proposition *unsafe* represent the property that node *N* is in an unsafe state. We can then express the desired property in CTL as follows:

$$\mathbf{AG}(\neg unsafe)$$

which says that for all states reachable from the set of initial states it is true that we never reach a state where unsafe is true. The negation of the property is

$$\mathbf{EF}(\neg unsafe)$$

which is true if there exists a state reachable from the initial state where unsafe is true; in other words if the network starts in one of the initial states it is possible to reach an unsafe state. The atomic proposition unsafe can stand for a property as complex as we desire. It could mean that a certain critical node has entered an undesirable state (e.g., a critical valve is open in a nuclear power plant), or it could mean that a certain unauthorized operation occurred at a critical node. For example, if a node represents a computer protecting a critical resource, it could represent the fact that somebody without the appropriate authority has logged onto the computer. The precise nature of a faulty state depends on the example at hand.

Service Related Properties

Many networked systems are built for distributed applications. For these cases we want to make sure that if a node *N* issues a service, then the service eventually finishes executing. Let the atomic proposition *start* express that a service was started, and *finished* express that the transaction is finished. The temporal logic formula given below expresses that *for all states where a service starts and all paths starting from that state there exists a state where the service always finishes,* or in other words *a service issued always eventually finishes*.

$$\mathbf{AG}(start \rightarrow \mathbf{AF}(finished))$$

For the banking example, we would like to verify that a check issued is always eventually cleared. This can be expressed in *CTL* as:

$$\mathbf{AG}(checkIssued \rightarrow \mathbf{AF}(checkCleared))$$

We can also analyze the effect of a compromised node (say *N).* Suppose we have modeled the effect of a malicious attack on node *N* (see discussion on injecting faults). Now we can check whether the desired properties are true in the modified networked system. If the property turns out to be true, the network is resistant to the malicious attack on the node *N.* This type of analysis is useful in determining vulnerable or critical nodes of a network with respect to a certain service. Using this analysis, if a node is found to be vulnerable or critical for a given service to complete, then the system administrator can deploy sophisticated intrusion detection algorithms for that node or bolster the security infrastructure around it. Thus our analysis can help identify the critical nodes in a networked system and therefore help determine whether it is survivable with respect to desired properties of a given service.

5.4 Step 4: Generate Scenario Graphs

We automatically construct scenario graphs via model checking. When a specified property is not true in a given model, a model checker will produce a counterexample, i.e., a trace or a scenario that leads to a final state that does not satisfy the property. (Details of model checking, e.g., see [5], are not needed to understand our method.) We exploit this functionality of model checkers to generate scenario graphs; i.e., a scenario graph is a compact representation of all the traces that are counterexamples of a given property [1]. For example, suppose we want to check whether during the execution of a networked system a certain event (e.g., buffer overflow~ never happens. If the property is not true (i.e., buffer overflow can happen), the scenario graph encapsulates all sequences of states and transitions that lead the system to a state where a buffer overflow occurs.

Scenario graphs depict ways in which a network can enter an unsafe state or ways in which a service can fail to finish. Scenario graphs encapsulate the effect of local faults on the global behavior of the network. If the architect models malicious attacks, the scenario graph is a compact representation of all the threat scenarios of the network, i.e., a set of sequences of intruder actions that lead the network to an unsafe state.

Fault Scenario Graphs

Recall that we can express the property of the absence of an unsafe reachable state as:

$$\mathbf{AG}(\neg unsafe)$$

If this formula is not true, it means that there are states that are reachable from the initial state that are faulty.

We briefly describe the construction of a scenario graph. Assume that we are trying to verify using model checking whether the specification of the network satisfies **AG**(¬*unsafe*). Usually, the first step in model checking is to determine the set of states S_r that are reachable from the initial state. After having determined the set of reachable states, the algorithm determines the set of reachable states S_{unsafe} that have a path to an unsafe state. The set of states S_{unsafe} is computed using fix-point equations [5]. Let R be the transition relation of the network, i.e., (s, *s'*) Є R iff there is a transition from state s to s' in the network. By restricting the domain and range of R to S_{unsafe} we obtain a transition relation R_f that encapsulates the edges of the scenario graph. Therefore, the scenario graph is $G = (S_{unsafe}, R_f)$, where S_{unsafe} and R_f represent the nodes and edges of the graph respectively. In symbolic model checkers, like NuSMV, the transition relation and sets of states are represented using *binary decision diagrams* (BDDs) [4], a compact representation for boolean functions. All the operations described above can be easily performed using BDDs. The BDD for the transition relation R_f is a succinct representation of the edges of the fault scenario graph. Since BDDs are capable of representing a large number of nodes, very large scenario graphs can be computed using our method.

Service Success/Fail Scenario Graphs

In the case of services, we are interested in verifying that every service started always eventually finishes. Recall that we express this property in *CTL* as:

$$\mathbf{AG}(start \rightarrow \mathbf{AF}(finished))$$

Since we allow several nodes to be faulty, in our experience we find that most of the time this property fails to hold. Thus more interestingly, during the model checking procedure, we derive two graphs: *a service success scenario graph* and *a service fail scenario graph.* The success scenario graph captures all the traces in which the service finishes; the fail scenario graph, all the traces in which the service fails to finish. These scenario graphs are constructed using a procedure similar to the one described for the

fault scenario graphs.

In our banking example, issuing a check corresponds to the start of a service. The scenario graph shown in Figure 3 shows the effect of link failures on the check clearing service for a check issued with source address Bank-A and destination address Bank-C (the start event is labeled as issueCheck(Bank-A,BankC) in the figure). The event corresponding to sending a check from location L1 to L2 is denoted as sendCheck(L1,L2). The predicates up(Link-A-2) and down(Link-A-2) indicate whether Link-A-2 is up or down. Recall that we allow links to fail nondeterministically. Therefore, an event sendCheck(Bank-A,MC2) is performed only if Link-A-2 is up, i.e., up(Link-A-2) is the pre-condition for event sendCheck(Bank-A,MC-2). If a pre-condition is not shown, it is assumed to be true. Note that a fault in a link can also be construed as an intruder taking over the link and shutting it down. From the graph it is easy to see that a check clears if Link-A-2 and Link-C-3 are up, or if Link-A-2 is down and Link-A-1 and Link-C-3 are up. We modified the model checker NuSMV to produce such scenario graphs automatically.

For realistic examples scenario graphs can be extremely large. Therefore, it is not feasible to enumerate all the scenarios or traces corresponding to a scenario graph. We developed a querying process by which an architect can select a subset of scenarios. First an architect identifies events of interest in the network; then, using these events as alphabet symbols, the architect provides a regular expression to specify the traces of interest. Consider the scenario graph shown in Figure 3 and this regular expression for the alphabet $\sum$:

$$\sum^* \text{ sendCheck(FRB-2,FRB-3) } \sum^*$$

This query captures the architect's interest in all traces where the check is transferred from FRB-2 to FRB-3, as denoted by the event sendCheck(FRB2, FRB-3). A trace that satisfies the regular expression is shown by a dotted line in Figure 3.

5.5 *Step 5:* Reliability and Latency Analysis

Once we have generated scenario graphs, we can perform reliability and latency analysis. First, we need to incorporate probabilities of various events into a given scenario graph to produce an MDP; then using the MDP we compute reliability and latency by calculating the value function corresponding to the optimal policy.

We first explain this analysis using the banking example and then provide a formal explanation. Let the boolean state variable A1 indicate whether Link-A1 is up, so A1 corresponds to Link-A-1's being down. Analogously, A2 and C3 are the boolean variables corresponding to Link-A-2's and Link-C-3's being up. In general an event will be associated with a boolean variable and the negation of the variable will denote that the event did not occur; we will use the boolean variable and the event it represents synonymously, e.g., event A1 corresponds to Link-A-1's being up.

We now explain how we handle dependencies between events. Assume that event A*2* is dependent on A1 and there are no other dependencies. Let P *(*A*1)* and P *(*C*3)* both be *a a* where P *(*A*1)* and P *(*C*3)* are the probabilities of Link-A-1 and Link-C-3 being up. The probability of event A2 depends on the event A1, and we give its conditional probability as:

$$P(A2 \mid A1) = 1/2$$
$$P(A2 \mid \bar{A}1) = 1/4$$

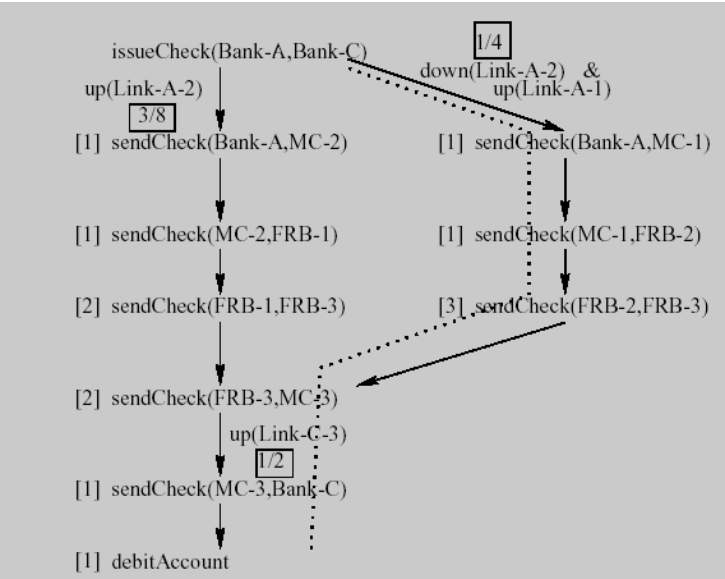


Figure 3: A Simple Scenario Graph

Reflect that if Link A-1 is down, it is more likely that Link A-2 will go down. In general, if an event A depends on the set of events $\{A_1, ..., A_k\}$, then the probability of A has to specified for each possible case in the set of events $\{A_1, ... , A_k\}$. For example, if A depends on $\{A_1 , A_2\}$, then $P(A|A_1 \wedge A_2)$, $P(A|A_1 \wedge \bar{A}_2)$, $P(A| \bar{A}_1 \wedge A_2)$, $P(A|A_1 \wedge A_2)$, and $P(A| \bar{A}_1 \wedge \bar{A}_2)$ have to be specified. This technique is the Bayesian network formalism.

In our example, first we have to compute the probability of the two events $A2$ and $\bar{A}2 \wedge A1$. These events correspond to events up(Link-A-2) and down(LinkA-2) & up(Link-A-1) in the scenario graph. The probabilities for these events are derived below.

$$P(A2) = P(A2|\bar{A}1)P(\bar{A}1) + P(A2|A1)P(A1)$$
$$= \tfrac{1}{4}(1-1/2) + \tfrac{1}{2} + \tfrac{1}{2}$$
$$= 3/8$$

$$P(\bar{A}2 \wedge A1) = P(\bar{A}2|A1)P(A1)$$
$$= (1 - P(A2|A1)P(A1))$$
$$= 1/4$$

We add these probabilities (shown inside little boxes) to the relevant edges of the scenario graph in Figure 3. Since we might assign probabilities to only some events (typically faults) and not others, we obtain a structure that has a combination of purely nondeterministic and probabilistic transitions. In our banking example, the architect might assign probabilities only to events corresponding to faults; the user of the banking system still nondeterministically issues checks. Intuitively, nondeterministic transitions are actions of the environment or the user, and probabilistic transitions correspond to moves of the adversary. If we view nondeterministic transitions as actions, the structure obtained after incorporating probabilities into the scenario graph is an MDP. (In the distributed algorithms literature [12], structures that have a combination of nondeterministic and probabilistic transitions are called *concurrent probabilistic systems.)*

We now explain the algorithm to compute reliability and latency by first considering a property about services. Recall that we are interested in the following property:

***AG**(start → **AF**(finished))*

Let *G* be the service success scenario graph corresponding to this property. Suppose each edge $s \rightarrow s'$ in *G* has a cost $c(s \rightarrow s')$ associated with it. Now the goal of the environment, which is assumed to be malicious, is to devise an optimal policy or equivalently choose nondeterministic transitions in order to minimize reliability or maximize latency. A value function *V* assigns a value *V(s)* for each state *s* in the scenario graph. Next we describe an algorithm to compute the value function *V* * corresponding to this optimal policy. This algorithm is called *policy iteration* in the MDP literature. (Later we explain how the value function can be interpreted as worst case reliability or latency. In the initial step, *V(s)* = 1 for all the states that satisfy the property *finished,* and for all other states *s* we assume that *V(s) = 0.* A state *s* is called *probabilistic* if transitions from that state are probabilistic. A state is called *nondeterministic* if it is not probabilistic. For all states *s* that satisfy *finished* the value *V(s)* is always 1; and for all other states the value function is updated as follows:

- If *s* is nondeterministic then

$$V(s) = \min_{s' \in succ(s)} c(c \rightarrow s') + V(s')$$

- If *s* is probabilistic then

$$V(s) = \sum_{s' \in succ(s)} p(s, s') \, (c(s \rightarrow s') + V(s'))$$

In the equations given above, *succ(s)* is the set of successors of state *s* and *p(s, s')* is the probability of a transition from state *s* to *s'*. Intuitively speaking, a nondeterministic move corresponds to the environment choosing an action to minimize the value. The value of a probabilistic state is the expected value of the value of its successors. Starting from the initial state, the value function *V* is updated according to the equations given above until convergence.

After the above algorithm converges, we end up with the desired value function *V* *. Let s_o be the initial state of the scenario graph.

• If the cost, *c*, associated with the edges is *zero,* then *V *(so)* is the *worst case reliability metric* corresponding to the given property, i.e., the worst case probability that if a service is issued it will eventually finish.
• If the cost, *c*, associated with the edges correspond to negative of the latency, then the value *-V *(so)* corresponds to the *worst case latency of* the service, i.e., the worst case expected finishing time of a service. Notice that in this setting, minimizing cost corresponds to maximizing latency.

Consider the scenario graph shown in Figure 3. The worst case reliability using our algorithm is (1/2 x 3/8) + (1/2 x 1/4) = 5/16. That is, the worst case probability that a check issued by Bank-A on Bank-C is cleared is 5/16. Latency in days for all the events is shown in Figure 3 inside square brackets, e.g., latency of the event sendCheck(FRB-3, MC-3) is 2 days. The worst case latency using our algorithm computes to be 4 days.

5.6 *Step 6:* Cost-Benefit Analysis

Finally, we add more cost information and extend our MDP to a CMDP. Again, we will explain this analysis using the running example first. Suppose an architect wants to upgrade some links to improve the overall robustness of the system. Three links Link-A-1, Link-A-2, and Link-C-3 are candidates for being upgraded. Assume that if Link-A-1 and Link-C-3 are upgraded then the probabilities *P(A1)* and *P(C3)* increase to 3/4 respectively. If Link-A-2 is upgraded then the probability of Link-A-2 being up is given below.

$$P(A2 \mid A1) = 3/4$$

$$P(A2 \mid \bar{A}1) = 3/8$$

If the links are not upgraded, then the probabilities do not change. In addition to the actions corresponding to the nondeterministic transitions, three extra actions (corresponding to upgrading Link-A-1, Link-A-2, and Link-C-3) are added to the action set, *A*, of the MDP that was constructed previously. Moreover, assume that the architect has a cost constraint so that only two links can be upgraded. Therefore, in this case we obtain a CMDP, where the cost of upgrading the links is expressed by the cost function *d* (Section 2). Algorithms for finding optimal policies in the case of *CMDPs* exist but are complicated [2]. Fortunately, our problem is easier because the decisions to upgrade the links are static, i.e., do not depend on the state of the system. In this case the optimal decision can be found by solving an auxiliary integer programming problem. With each of the three links Link-A-1, Link-A-2, and Link-C-3 we associate 0-1 variables X_{A1}, X_{A2} and X_{C3}. Intuitively, $X_{A1} = 1$ indicates that Link-A-1 has been upgraded. Now the worst case reliability is a function of *XA1, XA2,* and *XC3*. We denote this by $Rel(X_{A1}, X_{A2}, X_{C3})$. Our aim is to maximize the worst case reliability $Rel(X_{A1}, X_{A2}, X_{C3})$ subject to the constraint that at most two links can be upgraded, i.e.,

$$X_{A1}+X_{A2}+X_{C3} \leq 2$$

This is a non-linear integer programming problem. Although the problem in its full generality is hard, several heuristics for solving these class of problems have been studied [16]. For our example, Figure 4 lists the worst-case reliability for the three possible cases. It is clear that the best option is to upgrade Link-A-1 and Link-C-3.

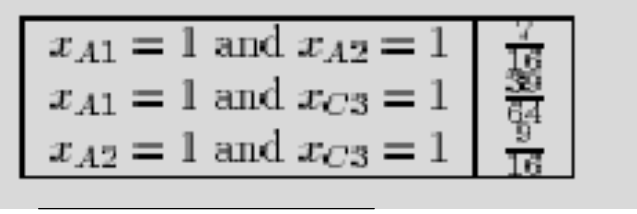


Figure 4: Table of Three Cases

6 Status

We built a tool *Trishul* based on the ideas presented in this paper. We implemented all the basic algorithms. We are finishing the graph visualization component and a customized editor.

We also finished two major case studies: an extended banking system and a bond trading floor. Our model of the banking system is much more complicated than the simplified example presented in this paper. For example, we handle protocols such as *Fedwire* and *SWIFT* (used for transfer of funds and transmitting financial messages respectively) that we did not show here. The entire banking system model is about 2,000 lines of NuSMV code. The scenario graph has about 25,000 nodes and computing reliability and latency takes only a few minutes.

We also modeled and analyzed the system architecture of a bond trading floor of a major investment company in New York. The model is about 10,000 lines of NuSMV code and has about 100 state variables. Our tool found several attacks. Two of these attacks were considered serious by the

architects. One attack enabled a junior trader to acquire a head trader's password. The second attack enabled a junior trader to obtain sensitive information from the company's database, i.e., a junior trader could find out the nature of the pending trades. Not surprisingly, we gained valuable experience during this case study. The most cumbersome part of the modeling process was the fault injection phase because the nature of the faults injected was heavily dependent on the security policies and technologies deployed at that node. We plan to automate the fault injection process in the near future.

7 Related Work

Survivability is a fairly new discipline, and viewed by many as distinct from the traditional areas of security and fault-tolerance [7]. The Software Engineering Institute uses a method for analyzing the survivability of network architectures (called SNA) and conducted a case study on a system for medical information management [8]. The SNA methodology is informal and meant to provide general recommendations of "best practices" to an organization on how to make their systems more secure or more reliable. In contrast, our method is formal and leverages off automatic verification techniques such as model checking. Other papers on survivability can be found in the *Proceedings of the Information Survivability Workshop [10]*.

Research on *operational security by* Ortolo, Deswarte, and Kaaniche [13] is closest to Step 4 of our method. Their attack state graphs are similar to our scenario graphs. However, since we use symbolic model checking to generate scenario graphs, represented by BDDs, we can handle extremely large graphs. Moreover, in our method a scenario graph corresponds to a particular service; in contrast their graph corresponds to a global model of the entire system. We are currently investigating how to incorporate concepts and analysis techniques presented in their paper [13]. into our method.

Fault injection is a well-known technique in the fault tolerance community. We allow the designer to specify any kind of fault, and thus we can consider a wider class of faults. Moreover, we allow fault events to be dependent and thus can model correlated attacks. Computing reliability is also not new. There is a vast amount of literature on verifying probabilistic systems and our algorithm for computing reliability draws on this work [6]. The novelty in our work is the systematic combination of different techniques into one method.

8 Summary of Contributions and Future Work

Survivability has become increasingly important with society's increased dependence on critical infrastructures run by computers. In this paper, we presented in a single framework a systematic method for analyzing a networked system for survivability. A fundamental contribution of our work is to use constrained Markov Decision Processes as the sole underlying mathematical model for this framework. A second contribution is the natural integration of a set of analysis techniques from disparate communities into this framework: model checking (popular in computer-aided verification), Bayesian network analysis (popular in artificial intelligence), probabilistic analysis (popular in hybrid systems and queueing systems), and cost-benefit analysis (popular in decision theory). In combination, these techniques let us provide a multi-faceted view of the networked system. This *holistic view* of a system is at the core of achieving survivability for the system's critical services.

There are several directions for future work. First, we plan to finish the prototype tool that supports our method. We are working on several case studies, including protocols used in an electronic commerce system. Since for real systems, scenario graphs can be very large, we plan to improve the display and query capabilities of our tool so architects can more easily manipulate its output. Finally, to make the fault injection process systematic, we are investigating how best to integrate operational security analysis tools such as COPS [9] into our method.

References

[1] Nusmv: a new symbolic model checker. http://afrodite.itc.it:1024/ nusmv/.

[2] E. Altman. *Constrained Markov Decision Processes.* Chapman and Hall, 1998.

[3] D. Bertsekas. *Dynamic Programming and Optimal Control.* Athena Scientific, 1995.

[4] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.,* C-35(8):677-691, Aug. 1986.

[5] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[6] C. Courcoubetis and M. Yannakakis. The complexity of probabilistic verification. *Journal of ACM,* 42(4):857-907, 1995.

[7] R. Ellison, D. Fisher, R. Linger, H. Lipson, T. Longstaff, and N. Mead. Survivable network systems: An emerging discpline. Technical Report CMU/SEI-97-153, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213, November 1997.

[8] R. Ellison, R. Linger, T. Longstaff, and N. Mead. Survivability network system analysis: A case study. *IEEE Software,* 16/4, July/August 1999.

[9] D. Farmer and E. Spafford. The cops security checker system. In *Proceedings Summer Usenix Conference,* 1990.

[10] In *Information Survivability Workshop, ISW,* October 1998. http://www.cert.org/research/isw98.html.

[11] J. Knight, M. Elder, J. Flinn, and P. Marx. Summaries of three critical infrastructure applications. Technical Report CS-97-27, Department of Computer Science, University of Virginia, Charlottesville, VA 22903, December 1997.

[12] N. Lynch, I. Saias, and R. Segala. Proving time bounds for randomized distributed algorithms. In *Proceedings PODC,* pages 314-323, 1994.

[13] R. Ortalo, Y. Deswarte, and M. Kaaniche. Experimenting with quantitative evaluation tools for monitoring operational security. *IEEE Transactions on Software Engineering,* 25/5:633-650, Sept/Oct 1999.

[14] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[15] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Comput.* Sci., 13:45-60, 1981.

[16] A. Schrijver. *Theory of Linear and Integer Programming.* John Wiley & Sons, 1998.

Survivable Systems

Tom Longstaff

Jeanette Wing

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Mission Survivability

Survivability

- What if
  - a terrorist hacker brings down the nation's power grid?
  - an act of Mother Nature causes the US banking network to fail?
- Critical infrastructures
  - Utilities: gas, electricity, nuclear, water, …
  - Communications: telephone, networks, …
  - Financial: banking, trading, …
  - Medical: emergency services, hospitals, …

3

Survivability

- A system is survivable if it can continue to provide end services despite the presence of faults.
- Faults
  - Accidental or malicious
  - Not necessarily independent

  ⇒ Finer-grained reliability analysis is enabled/required (and more relevant).
- Service-oriented
  - Exploit semantics of application

  ⇒ Not all network nodes and links are treated equally.

4

Foundational Questions

- What is the difference in models for survivability and those for
  - Fault-tolerant distributed systems?
  - Secure systems?
- Our starting point:
  - Independence assumption goes out the window.
  - Cost must be included in the equation.

5

Key properties

- Mission Focus
  - Identification of risks and trade-offs
  - Alternative means to meet mission
- Assume imperfect defenses

6

2 Parts in Cooperation

- The Survivable Network Architecture Method
  - Measures existing systems for survivability
  - Focuses on user and intruder models
- Inverting Formal Methods Techniques for Survivability
  - Applies model checking and other techniques to survivability
  - Allows systems that are formally specified to submit to survivability analysis

7

The Survivable Network Analysis Method

- Focus
  - early phase of life cycle
  - applications as well as system infrastructure
  - tailorable depending on stage of development.
- Three options for SNA analysis
  - survivability architecture
  - survivability requirements
  - mission lifecycle

8

Architectural Focus

- Capture assumptions such as boundaries and users
- Support system evolution as requirements and technologies change
  - evolving functional requirements
  - trend to loosely coupled
  - requirements for integration across diverse systems
- Assist with product selection and integration with respect to rapidly changing security product world

9

General Method

- Identify essential services with normal usage.
- Generate intrusion scenarios which are use cases for intruder
- Evaluate system in terms of response to scenarios
  - Requirements: propose response to intrusions
  - Architecture: evaluate system and operational behavior
- Mission impact
  - applications as well as system components
  - stakeholders input essential

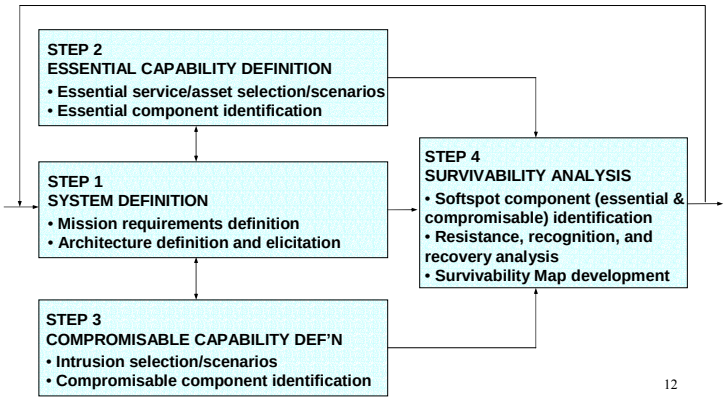
10

Survivability Architecture

- Make recommendations for survivability improvements
- Identify decision and tradeoff points - areas of high risk
- Identify trade-offs with other software quality attributes
  - safety, reliability, performance, usability

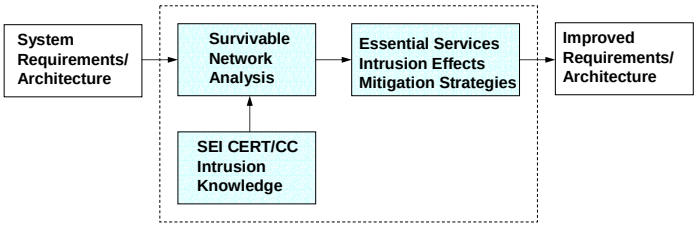
11

The Survivable Network Analysis Method



12

Determining Survivability Strategies



13

Survivability Map

Intrusion Scenario	Softspot Effects	Architecture Strategies for →	Resistance	Recognition	Recovery
(Scenario 1) …		Current			
		Recommended			
(Scenario n)		Current			
		Recommended			

- Roadmap for management evaluation and action

14

Option: Survivability Requirements

- Identify requirements for mission-critical functionality
  - minimum essential services
  - graceful degradation of services
  - restoration of full services
- Identify explicit requirements for
  - recovery
  - recognition
  - resistance

15

Option: Mission Lifecycle

- Factor survivability into the development and operational lifecycle
- Capture security and survivability assumptions
  - boundaries, users
- Identify survivability decision points
  - impact of changes on recovery, intrusion detection, etc.

16

Benefits of the SNA

- Clarified requirements
- Documented basis for system decisions
- Basis to evaluate changes in architecture
- Early problem identification
- Increased stakeholder communication

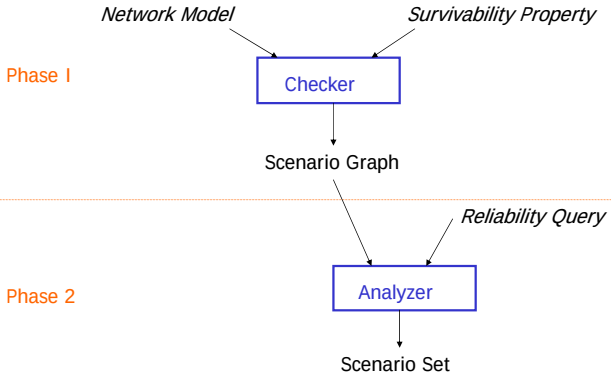
17

Additional Information

- SNA Case Study: The Vigilant Healthcare System
  - IEEE Software: July/August 1999
- Survivability: Protection Your Critical Systems
  - IEEE Internet Computing: Nov/December 1999
- Web site: IEEE article and other reports
  www.sei.cmu.edu/organization/programs/nss/surv-net-tech.html

18

Overview of Formal Method

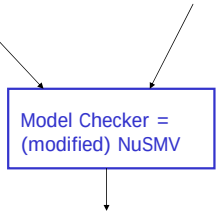


19

Phase 1

Network Model =
A set of concurrently executing Finite State Machines.

Survivability Property =
A predicate in CTL.



Scenario Graph =
A set of related examples.

20

Network Model

- Processes
  - Nodes and links are processes (i.e., FSMs)
    - banks, money centers, federal reserve banks, and links
  - Communication via shared variables (i.e., finite queues)
    - representing channels, and hence interconnections.
- Failures
  - Faults represented by special state variable
    - fault:{normal, failed, intruded}
  - Links and banks can fail at any time
    - Failed link blocks all traffic.
    - Failed bank routes all checks to an arbitrarily chosen money center.
  - Money centers and federal reserve banks do not fail.

21

Survivability Properties

- Fault-related
  - Money never deposited into wrong account.
    - $AG(\neg error)$
- Service-related
  - A check issued eventually clears.
    - $AG(checkIssued \rightarrow AF(checkCleared))$

22

Inputs to Model Checker

- State machines

  MODULE main

  fault: {normal, fail-stop, Byzantine, hacker-attack, terrorist-attack, link-down, …}

  …

  next (fault) := case

  fault = normal : {normal, fail-stop, …}

  …

  $P_i(v_n)$: {hacker-attack, terrorist-attack}

  default : fault

  esac

  MODULE bank(user, <other input parameters>)

  next (…) := case

  $P_j(v_m)$ & fault = normal => <route check to user.destination>

  ...

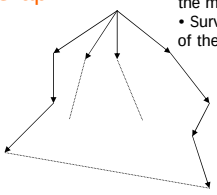
- Property

  AG not(faulty)

23

Output From Model Checker

- Fault Scenario Graph

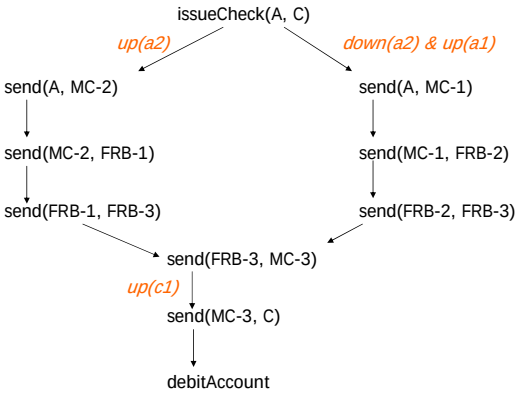


Intuition:
• Each "counterexample" spit out by the model checker is a scenario.
• Survivability property gives a slice of the model.

Each path is a scenario of how a fault can occur.

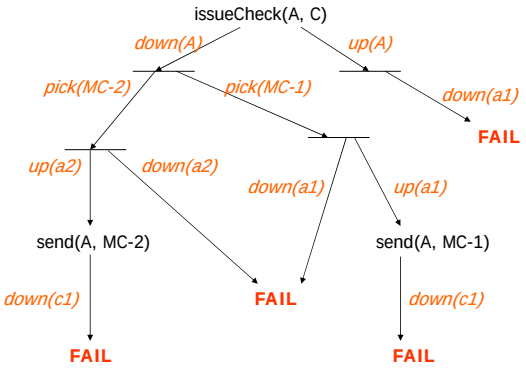
24

A *Service Success* Scenario Graph



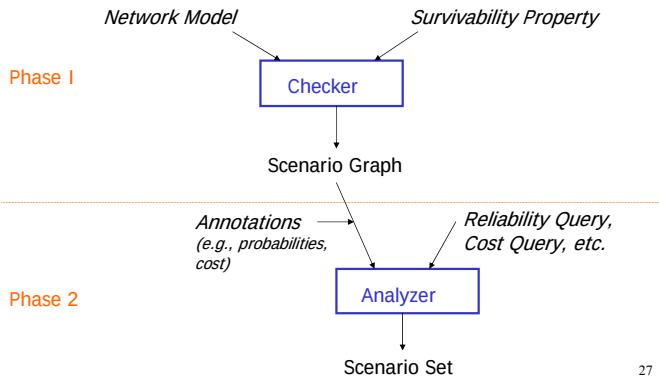
25

A *Service Fail* Scenario Graph



26

Overview of Method



27

Phase 2: Reliability Analysis

- Annotations = Probabilities
  - Use Bayesian Networks to model dependence of events.
- Symbolic
  - Use symbolic probabilities
    - high, medium, low
  - Use NDFA theory to compute scenario set.
- Continuous
  - Use numeric probabilities
    - [0.0, 1.0]
  - Use Markov Decision Processes to model both nondeterministic and probabilistic transitions.

28

State of Project

- Tools
  - Trishul tool
    - Uses NuSMV model checker, done by Somesh Jha
  - New tool
    - Uses SPIN, ongoing by Oleg Sheyner
- Case studies (Jha, Sheyner)
  - Trading floor model of major investment bank (being "sanitized" by Jha)
    - 10K lines of NuSMV
    - half-million nodes in scenario graph
    - 50 threat scenarios
    - 45 found by system
    - 5 new threat scenarios found
    - With independence assumption, too many misses.
  - B2B e-commerce NYC start-up (Jha)
    - 50K lines of Statecharts, 2 million NuSMV beyond capability of tool

29

Sample Open Research Questions

- Foundational
  - What is an appropriate fault-model for survivable systems?
    - Malicious attack versus Byzantine failure
  - What role does "service-oriented" really play in the notion of survivability?
  - What is an appropriate logic for describing survivability properties?
    - What logic or subset of CTL corresponds to finite scenario graphs?
- Pragmatic
  - How applicable is the CMDP model for other critical infrastructure examples?
    - How far can we push the analysis techniques?
  - What combination of tools can further automate the analysis?
    - Linear programming packages, theorem provers, …
  - How can you design a system for survivability?

30